

Giovanni Pighizzini
Mauro Ferrari

Dai fondamenti agli oggetti

Corso di programmazione Java

Terza edizione



3.3 Il tipo primitivo <code>boolean</code>	84
3.4 Operatori booleani e condizioni	85
3.5 I cicli <code>while</code> e <code>do...while</code>	89
3.6 L'istruzione <code>for</code>	96
3.7 Le istruzioni <code>break</code> e <code>continue</code>	103
4 Tipi primitivi e tipi enumerativi	115
4.1 Espressioni	115
4.2 Riepilogo degli operatori	121
4.3 Tipi numerici interi	124
4.4 Tipi numerici in virgola mobile	125
4.5 Conversioni implicite ed esplicite di tipo	128
4.6 Il tipo <code>char</code>	135
4.7 I tipi enumerativi	142
4.8 L'istruzione <code>switch</code>	145
4.9 I metodi statici	151
4.10 Le classi involucro	159
5 Array e collezioni	165
5.1 Array di oggetti	165
5.2 Array e cicli <code>for</code>	176
5.3 Il parametro del metodo <code>main</code>	177
5.4 Array di tipo primitivo	178
5.5 Array e tipi enumerativi	184
5.6 Array di array	190
5.7 La classe <code>Sequenza</code> : introduzione ai tipi generici	196
6 Uso della gerarchia	207
6.1 La classe <code>Rettangolo</code>	209
6.2 UML: rappresentazione di classi e oggetti	218
6.3 La classe <code>Quadrato</code>	219
6.4 Ereditarietà e polimorfismo	223
6.5 Le classi astratte	229
6.6 La gerarchia delle classi	237
6.7 Gerarchia e uso dei riferimenti	238
6.8 Scelta del metodo da eseguire	239
6.9 Esempio: gestione di un elenco di figure	245
6.10 I file di testo	250
6.11 Esempio: la tavola delle occorrenze	255
6.12 Le interfacce Java	260
6.13 L'interfaccia <code>Iterable</code> e il ciclo <code>for-each</code>	263
6.14 La gerarchia dei tipi	266
6.15 Gerarchia dei tipi e tipi generici	271

6.16	Vincoli sui segnaposto	279
6.17	Tipi generici e vincoli sugli argomenti	284
6.18	Tipi generici e compilazione	285
II	Implementazione degli oggetti	287
7	Implementazione delle classi	289
7.1	Classi e oggetti	289
7.2	La classe Frazione : alcuni miglioramenti	300
7.3	Una nuova implementazione della classe Frazione	303
7.4	Esempio: la classe Orario	309
7.5	I campi e i metodi statici	316
7.6	Riepilogo della struttura delle classi	322
7.7	Implementazione di un'interfaccia	327
7.8	Documentazione delle classi	329
7.9	I package	337
7.10	I modificatori di visibilità public e “amichevole”	342
7.11	Documentazione dei package	346
7.12	UML: membri di una classe	347
8	Estensione delle classi	349
8.1	Ereditarietà e implementazione di sottoclassi	349
8.2	Costruttori e gerarchia delle classi	352
8.3	Il riferimento super	355
8.4	Ereditarietà e stato degli oggetti	356
8.5	Overloading dei metodi	362
8.6	Overriding, overloading e scelta del metodo da eseguire	363
8.7	Il metodo equals	370
8.8	Variabili e adombramento	375
8.9	Esempio	378
8.10	Implementazione della classe Figura	382
8.11	Implementazione della classe Rettangolo	383
8.12	Nuovi metodi per le classi Rettangolo e Quadrato	386
8.13	Il modificatore di visibilità protected	391
8.14	Il modificatore final	394
9	Tipi enumerativi, tipi generici e interfacce	401
9.1	Definizione di tipi enumerativi	401
9.2	Definizione di classi generiche	409
9.3	Metodi generici	413
9.4	Definizione di interfacce	416
9.5	Uso del supertipo definito dall'interfaccia	417

10 Organizzazione della memoria e ricorsione	427
10.1 Invocazione di metodi, passaggio di parametri e rientro	427
10.2 Organizzazione della memoria	431
10.3 Metodi ricorsivi	438
10.4 Metodi con un numero variabile di argomenti	450
11 Eccezioni	453
11.1 Le eccezioni	453
11.2 Come intercettare le eccezioni: l'istruzione try-catch	457
11.3 Rientro dai metodi ed eccezioni	460
11.4 Esempio: una calcolatrice in notazione postfissa	461
11.5 Alcune eccezioni	479
11.6 Come sollevare le eccezioni: l'istruzione throw	485
11.7 Come definire un'eccezione	490
11.8 Eccezioni controllate e non controllate	500
11.9 Perché le eccezioni controllate?	511
11.10 Gli errori	512
11.11 La clausola finally	513
11.12 Ridefinizione di metodi ed eccezioni	514
11.13 Metodi astratti ed eccezioni	517
III Argomenti avanzati	519
12 Strutture dati dinamiche	521
12.1 Implementazione di strutture a pila	521
12.2 Le code	530
12.3 Le liste ordinate	539
12.4 Alberi binari	554
12.5 Alberi di ricerca	556
12.6 Implementazione degli alberi di ricerca in Java	558
12.7 Esempio: elenco alfabetico di stringhe	565
12.8 Ulteriori esempi sulle liste	571
12.9 Ulteriori esempi sugli alberi	574
13 Gli stream	579
13.1 Stream di caratteri	580
13.2 Le gerarchie Reader e Writer	584
13.3 La classe File	590
13.4 Stream di byte	595
13.5 Lettura e scrittura di dati primitivi	596
13.6 La classe PrintStream	598
13.7 I flussi di input/output standard	598

13.8 Lettura e scrittura di oggetti su stream di byte	601
IV Appendici	609
A Tipi primitivi	611
A.1 Tipi interi	612
A.2 Il tipo char	613
A.3 Operatori	614
B Strutture di controllo	615
B.1 L'istruzione if-else	615
B.2 L'istruzione while	616
B.3 L'istruzione do...while	617
B.4 L'istruzione for	617
B.5 L'istruzione switch	619
B.6 Le istruzioni break e continue	620
B.7 L'istruzione return	621
B.8 L'istruzione try-catch	622
C Installazione di JDK 6 e del package prog	625
C.1 Installazione	625
C.2 Prima esecuzione	631
C.3 I package	634
C.4 Installazione del package prog	637
Note bibliografiche	639
Indice analitico	641

Prefazione

La programmazione consiste fondamentalmente nel codificare il procedimento risolutivo di un problema (cioè un *algoritmo*) in un insieme di istruzioni destinate a un esecutore, creando appunto il *programma*.

Agli inizi i programmi venivano scritti direttamente nel linguaggio comprensibile al processore cui erano destinati; con l'introduzione dei linguaggi di programmazione si è passati, via via, alla scrittura di programmi destinati a esecutori più evoluti, raggiungendo livelli di astrazione sempre più elevati. Tutto ciò ha modificato profondamente il modo di scrivere i programmi, al punto che, attualmente, gran parte dell'attività di programmazione consiste nel combinare tra loro "mattoni" già pronti. Questo, sebbene a prima vista possa apparire banale, richiede un maggiore livello di astrazione e di flessibilità. Il programmatore deve essere in grado di adattarsi a ciò che ha a disposizione: deve capire come sono fatti i mattoni che può utilizzare e deve avere l'abilità di combinarli in modo adeguato per ottenere la soluzione desiderata. Di volta in volta i mattoni possono cambiare: oltre che dalle librerie standard, ampiamente disponibili, essi possono provenire, ad esempio, da codice specializzato per il controllo di particolari dispositivi o per la soluzione di una determinata classe di problemi, oppure da codice scritto da colleghi nell'ambito di uno stesso progetto.

Un altro aspetto importante, strettamente legato all'evoluzione dei mezzi di calcolo, è che, in molti casi, lo svolgimento di un compito non dipende da un unico esecutore, ma è il risultato delle interazioni di diversi agenti, come del resto avviene nel mondo reale. Ad esempio, nella visualizzazione di una pagina web intervengono almeno due agenti: il browser, cioè il programma utilizzato per "navigare" nella rete, e il server su cui risiede la pagina. La visualizzazione della pagina è frutto della loro interazione (è facile osservare che, in realtà, gli agenti coinvolti in questa operazione sono molti di più).

Questo testo è un vero e proprio corso di programmazione, sviluppato facendo riferimento alle idee appena esposte. Il corso è accessibile anche a chi non ha alcuna conoscenza della materia, e nel contempo tocca e approfondisce le tematiche e gli aspetti fondamentali della programmazione.

Il paradigma a oggetti rappresenta l'ambiente ideale per avvicinarsi alla programmazione nell'ottica descritta sopra. Gli oggetti non sono altro che i mattoni base con cui costruire i programmi. La computazione è affidata a vari oggetti che possono lavorare autonomamente e cooperare tra loro. Ogni oggetto è in grado di fornire un insieme di servizi. Compito principale

del programmatore è quello di coordinare tra loro gli oggetti al fine di ottenere il comportamento desiderato. Pertanto, il testo fa ampio riferimento a questo paradigma, senza tuttavia trascurare i tradizionali argomenti relativi alla programmazione con linguaggi imperativi.

Per varie ragioni, tra cui la relativa semplicità e la vasta diffusione, abbiamo scelto di concretizzare i concetti presentati nel libro tramite il linguaggio Java. Tali concetti costituiscono comunque un bagaglio sicuramente utile al lettore per affrontare lo studio successivo di linguaggi differenti, sia imperativi sia a oggetti. Infatti, lo scopo del testo non è insegnare un linguaggio di programmazione, ma piuttosto insegnare la programmazione utilizzando come ausilio un linguaggio adeguato, nel nostro caso Java.

Tradizionalmente l'introduzione della programmazione a oggetti avviene senza insistere troppo sulla differenza tra l'uso delle classi e la loro implementazione. Quasi tutti i testi, fin dai primi capitoli, trattano subito l'implementazione delle classi. Questo approccio, che abbiamo sperimentato anche noi in passato, concentra l'attenzione principalmente sui dettagli implementativi, facendo perdere di vista i differenti livelli di astrazione. Di fronte a esercizi in cui si richiede l'uso di classi già esistenti, molti studenti tendono a riscrivere tutto *ex novo*, un po' come se per ascoltare la musica da una radio la si aprisse con un cacciavite. Da queste considerazioni è nata la scelta, che caratterizza e differenzia notevolmente questo testo dai numerosi altri passati per le nostre mani, di suddividere i contenuti in due parti fondamentali, la prima relativa all'uso, la seconda all'implementazione delle classi. Riteniamo che in questo modo, tenendo cioè separati i due aspetti e chiarendone i diversi livelli di astrazione, si riesca a enfatizzare la parte orientata agli oggetti senza trascurare le parti imperative.

Nella prima parte del testo, il lettore impara a individuare classi e oggetti utili alla soluzione di determinati problemi, e a combinarli tra loro usando quanto è già a disposizione nelle librerie. In pratica, il lettore imparerà a risolvere problemi, anche complessi, scrivendo brevi e semplici programmi che sfruttino il più possibile codice già esistente. Come evidenziato nel Sommario, vengono introdotti in questa parte concetti fondamentali, sia legati alla programmazione tradizionale sia a quella a oggetti, tra cui i concetti di variabile e tipo e le strutture di controllo fondamentali. L'aspetto particolarmente innovativo e originale del testo, cioè il livello di astrazione, è subito evidente: anziché limitarsi a mostrare come combinare le istruzioni base del linguaggio mediante le strutture di controllo, insistiamo fin dai primi esempi su come combinare i servizi forniti dalle classi e dagli oggetti. Questo ha il vantaggio di consentire, da subito, la codifica di programmi in grado di svolgere compiti complessi, ma soprattutto abitua immediatamente il lettore a pensare in astratto al comportamento degli oggetti, cioè a ciò che essi fanno, senza soffermarsi sui dettagli implementativi, rimandati alla parte successiva del testo. In questa prima parte introduciamo dal punto di vista dell'uso anche i concetti fondamentali della programmazione a oggetti, come l'ereditarietà, la gerarchia delle classi, le classi astratte e il polimorfismo. La loro presentazione risulta così assai semplificata.

La seconda parte del testo è destinata invece al progetto e all'implementazione delle classi e degli oggetti. Dopo avere acquisito, nella prima parte, l'abilità di utilizzare classi già pronte, il lettore imparerà a implementarne di nuove, che potranno essere utilizzate nei suoi programmi o in programmi scritti da altri. In questa parte vengono sviluppati in dettaglio svariati argomenti, tra cui i concetti di *overloading* e *overriding* e le interfacce. Un intero capitolo è inoltre dedicato

al meccanismo delle eccezioni, alla loro definizione, trattamento e uso per la stesura di codice semplice ed elegante che gestisca le situazioni anomale. Abbiamo scelto di inserire un capitolo relativo all'organizzazione della memoria durante l'esecuzione, argomento spesso trascurato, ma estremamente formativo in un corso di programmazione.

Il testo contiene anche una breve introduzione alla notazione UML (*Unified Modeling Language*) e diversi approfondimenti. In particolare, abbiamo riservato un intero capitolo alle strutture dati dinamiche e alla loro implementazione in Java. In un capitolo preliminare trattiamo l'evoluzione della programmazione dai linguaggi macchina ai linguaggi a oggetti passando per la programmazione strutturata.

I concetti sono illustrati per mezzo di numerosi esempi. Molto spesso l'estensione di un esempio è lo spunto per l'introduzione di un nuovo concetto. In ogni caso, dopo una prima esemplificazione, ogni concetto viene approfondito e studiato nei dettagli, ricorrendo, quando è necessario sottolineare particolari aspetti, a ulteriori esemplificazioni.

Poiché l'apprendimento della programmazione non è un'attività mnemonica, ma richiede lo sviluppo di abilità acquisibili solo con esperienza e applicazione, il testo comprende numerosi esercizi, indispensabili e insostituibili strumenti di apprendimento.

Il CD-ROM distribuito con il libro contiene alcune librerie da affiancare a quelle standard di Java, utilizzate principalmente per gli esempi e per gli esercizi, oltre al codice sorgente dei principali esempi. Il CD-ROM include anche l'ambiente Java Standard Edition Development Kit 6 di Sun Microsystems, per i sistemi operativi Windows e Linux.

Il testo è stato ideato per un corso universitario di programmazione accessibile a chi non conosce l'argomento. In particolare, i contenuti si sono evoluti e consolidati negli anni sulla base di un uso sperimentale di versioni preliminari nell'ambito dei corsi integrati di "Programmazione" e "Laboratorio di Programmazione" che teniamo presso i corsi di Laurea in Informatica dell'Università degli Studi di Milano e dell'Università degli Studi dell'Insubria.

Novità del linguaggio e novità del testo

Dalla sua nascita, a metà degli anni '90, l'ambiente Java ha subito una continua evoluzione che ha riguardato sia il linguaggio stesso, sia l'infrastruttura associata (librerie, Java Virtual Machine, ecc.).

Sicuramente, le novità più rilevanti per il linguaggio sono quelle introdotte a partire dalla versione 5, rilasciata da Sun Microsystems il 30 settembre 2004. Tra queste vi sono in particolare, i tipi enumerativi, l'autoboxing e unboxing, i tipi generici e il ciclo for-each. Java 6 (ultima versione disponibile nel momento in cui scriviamo), al contrario, non ha visto cambiamenti nel linguaggio, ma aggiornamenti nell'infrastruttura.

La seconda edizione del testo era nata dalla necessità di presentare le nuove importanti caratteristiche introdotte con Java 5. L'organizzazione generale e, soprattutto, la metodologia di esposizione degli argomenti sono rimaste quelle della prima edizione. I contenuti sono stati aggiornati integrandoli con le nuove caratteristiche del linguaggio. Nei casi più semplici, l'integrazione è avvenuta aggiungendo paragrafi *ad hoc*, più spesso abbiamo dovuto riscrivere alcune parti che, alla luce delle novità introdotte, risultavano obsolete. Nella terza edizione, allineata a

Java 6, il materiale è stato ulteriormente rivisto e raffinato: sono stati inseriti ulteriori esercizi, rielaborate e aggiornate le librerie utilizzate negli esempi e fornite nel CD-ROM allegato al testo.

Anche se i tipi generici non sono più una novità, come lo erano per l'edizione precedente, riteniamo utile sottolineare la loro importanza, non solo all'interno del linguaggio stesso, ma, e soprattutto, a livello concettuale, con le loro contrastanti caratteristiche di facilità e di difficoltà. I tipi generici permettono di realizzare, con estrema facilità e con una conoscenza elementare del linguaggio, programmi che manipolano strutture dati complesse. Pensiamo ad esempio a una struttura a pila. Nelle versioni di Java precedenti era disponibile la classe `Stack` per rappresentare pile di oggetti (istanze della classe `Object`). Per scrivere applicazioni che utilizzassero pile contenenti dati di un tipo particolare, ad esempio pile di `Integer`, era necessario conoscere i concetti fondamentali relativi all'ereditarietà e alla gerarchia e ricorrere all'operazione, molto spesso "mal digerita", di cast tra tipi riferimento.

Dalla versione 5 del linguaggio (al prezzo di una sintassi piuttosto barocca) è possibile costruire direttamente pile di `Integer`, scrivendo il codice che le utilizza senza necessità di conoscere i concetti relativi alla gerarchia delle classi e dei tipi. Questa caratteristica, che fornisce al linguaggio un ulteriore meccanismo di astrazione, costituisce una notevole semplificazione nell'uso del linguaggio e si accorda proprio alla filosofia di riuscire a svolgere compiti complessi con pochi strumenti. In quest'ottica abbiamo introdotto i tipi generici nel Capitolo 5.

Come tutti gli altri tipi riferimento, anche i tipi generici si collocano all'interno della gerarchia dei tipi. Inoltre, per alcuni di essi, il livello di genericità deve essere necessariamente limitato. Mentre, ad esempio, è facile pensare a una pila di oggetti di un qualunque tipo, per le sequenze ordinate dovremo limitarci a considerare oggetti tra loro confrontabili secondo una relazione d'ordine. Per modellare queste caratteristiche sono stati introdotti i segnaposto (wild-card) e i vincoli su di essi. Questi aspetti legati ai tipi generici risultano piuttosto complessi e sicuramente difficili per chi si avvicini per la prima volta al linguaggio. Per completezza e poiché ricorrono spesso nella documentazione, abbiamo deciso di includere nel testo (anche se non in maniera del tutto esaustiva) queste parti. Chi si avvicina per la prima volta al linguaggio potrà ometterle e prenderle in esame per approfondimenti successivi.

A scopo di completezza e per non essere legati allo specifico linguaggio, abbiamo ritenuto opportuno trattare anche gli array. Di fatto, con le nuove caratteristiche di Java, questo argomento, che appare sempre più legato alla programmazione imperativa, avrebbe potuto essere omesso.

Supporto web

L'home page del testo si trova all'indirizzo

<http://pighizzini.dico.unimi.it/jb>

Qui potrete trovare eventuale materiale integrativo o di aggiornamento, sviluppato successivamente alla pubblicazione del libro. Da questa pagina potete inoltre raggiungere le nostre home page e quelle dei nostri corsi, nelle quali è possibile reperire altro materiale.

Ringraziamenti

Questa pubblicazione è frutto della riclaborazione attraverso gli anni del materiale didattico preparato per gli studenti del nostro corso. Vogliamo quindi ringraziare i nostri studenti che, anche inconsapevolmente, hanno contribuito alla realizzazione di questo lavoro: il feedback positivo di molti è stato di grande incoraggiamento; le loro incertezze sono state uno stimolo a rivedere criticamente e a riscrivere alcune parti. Ringraziamo in particolare tutti coloro che hanno segnalato errori e imprecisioni.

Un ringraziamento a Pearson per avere accettato di pubblicare il nostro lavoro, rinnovando la fiducia nei nostri confronti, anche per questa terza edizione. In particolare, vogliamo ringraziare Alessandra Piccardo, academic editor, per l'entusiasmo con cui ha sostenuto fin dall'inizio e ha continuato a sostenere questo progetto, Micaela Guerra, per il costante e preciso supporto, e Bruno Lanata, copy-editor, per il paziente e puntuale lavoro di revisione del manoscritto grazie al quale idee e concetti hanno trovato la corretta espressione in un linguaggio semplice e scorrevole.

*Giovanni Pighizzini
Mauro Ferrari
Milano, gennaio 2008*

Computer, algoritmi e linguaggi

L'*informatica* può essere definita come la disciplina che si occupa dell'*informazione* e del suo trattamento in maniera *automatica*. Questa definizione evidenzia due elementi fondamentali: da un lato il concetto di *informazione*, dall'altro i *mezzi* per elaborarla. Come mezzi per l'elaborazione dell'informazione non intendiamo solo quelli fisici, come i computer, ma anche, e prima di tutto, i *procedimenti* di elaborazione, denominati *algoritmi*.

1.1 Algoritmi

Il concetto centrale dell'informatica è appunto quello di *algoritmo*, inteso come procedimento per la trasformazione delle informazioni. Lo studio degli algoritmi può essere svolto indipendentemente dagli strumenti fisici utilizzati per la manipolazione dell'informazione (i computer) e, in effetti, la sua nascita ha preceduto di molti secoli quella dei calcolatori.

In realtà la nozione di algoritmo non riguarda solo l'informatica e il trattamento dell'informazione, ma anche qualunque campo in cui si possano descrivere sequenze di operazioni, finalizzate allo svolgimento di un compito. Ad esempio, una ricetta di cucina è un algoritmo che può essere eseguito da un cuoco, mentre uno spartito musicale è un algoritmo eseguibile da un pianista.

Il primo esempio di algoritmo che presentiamo è inherente all'ambito matematico. Consideriamo il problema del calcolo del massimo comun divisore tra due interi positivi. Tale problema può essere risolto con il seguente procedimento, noto come *algoritmo di Euclide*, dal nome del matematico greco che l'ha scoperto.

1. Siano x e y i due numeri.
2. Calcola il resto della divisione di x per y .
3. Se il resto è diverso da zero, ricomincia dal passo 2
utilizzando come x il valore attuale di y , e come y il valore del resto,
altrimenti prosegui con il passo successivo.
4. Il massimo comun divisore è uguale al valore attuale di y .

Si osservi che chiunque sappia comprendere ed eseguire le operazioni che costituiscono l'algoritmo di Euclide può calcolare il massimo comun divisore tra due numeri senza conoscere il

significato dell'algoritmo stesso e, addirittura, senza sapere che cosa sia il massimo comun divisore tra due numeri. In altre parole, l'esecutore può svolgere il procedimento senza avere la minima idea di quale ne sia lo scopo (si può immaginare che egli si limiti a eseguire fedelmente gli ordini che gli vengono impartiti, senza chiedersi nulla circa il loro significato). Pertanto l'“intelligenza” necessaria per trovare la soluzione del problema (in questo caso per calcolare il massimo comun divisore) è tutta codificata nell'algoritmo.

Vediamo ora di definire con maggior precisione la nozione di algoritmo: un *algoritmo* è un insieme ordinato di passi *eseguibili* e *non ambigui*, che definiscono un processo che *termina*.

Osserviamo che nella definizione precedente si richiede che i passi che costituiscono un algoritmo siano *eseguibili* o, in altre parole, che l'algoritmo sia *effettivo*. Si considerino ad esempio le seguenti istruzioni.

1. Crea un elenco di tutti i numeri primi.
2. Ordina l'elenco in modo decrescente.
3. Preleva il primo elemento dall'elenco risultante.

Queste istruzioni non descrivono un algoritmo effettivo, in quanto la prima e la seconda istruzione non sono effettivamente eseguibili: richiedono infatti la manipolazione di infiniti elementi.

La seconda condizione presente nella definizione di algoritmo è che i passi siano *non ambigui*. Questo significa che l'azione da compiere a ogni passo dev'essere univocamente determinata e non permettere gradi di libertà.

Infine si richiede che il processo definito dall'algoritmo *termini*, cioè conduca alla soluzione in un numero finito di passi.

1.2 Programmi e linguaggi di programmazione

Abbiamo visto che un algoritmo indica un metodo, un procedimento, per la soluzione di un problema. Un *programma* è l'espressione di un algoritmo in un linguaggio che l'esecutore è in grado di comprendere *senza bisogno di ulteriori spiegazioni*. Pertanto, si può immaginare un algoritmo come un oggetto astratto, concettuale, mentre un programma ne è l'espressione concreta. Lo stesso algoritmo può essere espresso in differenti linguaggi, in base agli esecutori cui è destinato. La scrittura del programma costituisce quindi una fase successiva all'individuazione dell'algoritmo risolutivo di un determinato problema.

Per rappresentare concretamente un algoritmo si utilizzano dei blocchi di base, cioè delle istruzioni primitive, che l'esecutore è in grado di comprendere ed eseguire direttamente, e delle regole per combinare tra di loro le istruzioni primitive. L'insieme delle istruzioni primitive unite alle regole di combinazione costituisce la base dei *linguaggi di programmazione*. Ogni istruzione primitiva ha una *sintassi* e una *semantica*. La *sintassi* è la forma con cui l'istruzione primitiva viene espressa, la *semantica* è il significato dell'istruzione stessa. L'esecutore, riconoscendo un'istruzione in base alla sintassi, attua l'azione corrispondente alla semantica.

1.3 Informazione

Le entità su cui opera un computer (come i dati e i risultati) prendono il nome di *informazione*. L'informazione può essere misurata utilizzando un'unità detta *bit* (da *binary digit*). Intuitivamente un bit corrisponde alla quantità di informazione che otteniamo quando riceviamo la risposta a una domanda binaria, cioè a una domanda che ammette solo le due risposte "sì" e "no" (nell'ipotesi che le due risposte abbiano la stessa probabilità). In altri termini, con un bit d'informazione è possibile rappresentare uno tra due possibili valori, come giorno/notte, sole/luna, acceso/spento, falso/vero, 0/1, etc. Con due bit è possibile rappresentare quattro valori differenti (00, 01, 10, 11). In generale, con n bit è possibile rappresentare 2^n valori differenti.

Una sequenza di 8 bit viene detta *byte*. Pertanto un byte rappresenta un valore tra $256 = 2^8$ possibili valori o, in altre parole, un byte può assumere 256 valori differenti. I prefissi K (Kilo), M (Mega) e G (Giga) indicano alcuni multipli del byte: rispettivamente 2^{10} (un po' più di mille), 2^{20} (un po' più di un milione) e 2^{30} (un po' più di un miliardo) byte.

1.4 Computer e programmazione, hardware e software

Se volessimo definire un computer, potremmo dire che è una macchina elettronica *programmabile* per lo svolgimento di diverse funzioni. La caratteristica fondamentale che distingue un computer, ad esempio, da una calcolatrice, è appunto la sua *programmabilità*: avendo in ingresso un programma e dei dati, un computer produce risultati secondo il procedimento di trasformazione descritto dal programma.

Analogamente, possiamo definire un computer come una macchina in grado di eseguire un processo computazionale sulla base di regole specificate tramite un *programma*. Il programma è costruito a partire da istruzioni elementari che il computer è in grado di comprendere ed eseguire. L'insieme di tali istruzioni è generalmente piuttosto ristretto; la potenza e la vasta applicabilità dei computer è data dalla capacità di operare in maniera estremamente veloce e accurata. Combinando azioni elementari in lunghe sequenze di istruzioni è possibile far eseguire a un computer compiti complessi.

La *programmazione* è l'attività che consiste nell'organizzare istruzioni elementari, direttamente comprensibili dall'esecutore, in strutture complesse, i *programmi*, al fine di svolgere determinati compiti. Quando introdurremo il concetto di *compilatore* vedremo che, fortunatamente, non è necessario scrivere programmi nel linguaggio, estremamente povero, comprensibile dal processore di un computer, ma si possono utilizzare linguaggi ad alto livello, comprensibili da esecutori *astratti*.

In molti casi, la computazione non può essere modellata considerando semplicemente l'esecuzione, passo per passo, di un programma. Infatti, in un sistema complesso hanno un'importanza fondamentale le *interazioni* e le *comunicazioni* tra differenti componenti. Si consideri, ad esempio, un *browser* web. La visualizzazione di una pagina non è frutto solo dell'interazione tra il browser e l'utente, ma è anche il risultato di interazioni con altri elaboratori e programmi che possono trovarsi a notevole distanza. Un secondo aspetto della programmazione è pertanto quel-

lo di *coordinare* le interazioni tra diverse entità. La *programmazione a oggetti* è una metodologia che si presta molto bene a questo tipo di attività.

Con i termini *hardware* e *software* indichiamo le due “anime” di un computer: l’hardware è la parte fisica, costituita da un insieme di circuiti opportunamente connessi e da vari dispositivi, il software è l’insieme di tutti i programmi.

Hardware

La configurazione dell’hardware può variare notevolmente da un elaboratore all’altro, sia per quanto riguarda i dispositivi presenti sia per le loro caratteristiche. Senza entrare nei particolari, è tuttavia possibile riconoscere alcuni elementi fondamentali in comune.

- *Processore o unità centrale (CPU, Central Processing Unit).*

È la parte che esegue effettivamente l’elaborazione.

- *Memoria centrale.*

Contiene il programma (o i programmi) in esecuzione e i dati (o parte dei dati) su cui esso opera. La memoria centrale ha capacità limitata (oggi vengono venduti personal computer con una memoria centrale dell’ordine di qualche Gigabyte), ma permette di reperire i dati abbastanza velocemente. Infatti la memoria centrale è una memoria ad accesso diretto (detta anche RAM, *Random Access Memory*), cioè una memoria nella quale un dato può essere reperito conoscendone la posizione. La memoria RAM è una memoria a lettura e scrittura, nella quale è cioè possibile leggere o scrivere dati. L’informazione nella memoria RAM viene mantenuta solo in presenza di alimentazione. La memoria centrale di un computer contiene di solito anche un’area di dimensioni piuttosto limitate detta ROM (*Read Only Memory*), a sola lettura, che mantiene l’informazione anche in assenza di alimentazione. Quest’area contiene informazioni necessarie all’avvio del computer.

La memoria centrale di un elaboratore è suddivisa in *celle* (dette anche *parole* o *locazioni*) tutte della stessa dimensione (ad esempio di 4 byte). Ogni cella è identificabile tramite un numero, detto *indirizzo*. Il processore può effettuare un’operazione di lettura o scrittura in una cella di memoria specificandone l’indirizzo. Si osservi che, quando si effettua la scrittura di un dato in una cella di memoria, il valore che essa conteneva precedentemente viene perso. Il tempo d’accesso a una cella è costante e indipendente dalla sua posizione.

- *Memoria di massa.*

Al contrario della memoria centrale, la memoria di massa è in grado di memorizzare grandi quantità di informazione (gli hard disk dei personal computer odierni hanno capacità di diverse centinaia di Gigabyte). Inoltre l’informazione viene memorizzata in maniera permanente, cioè non viene persa in mancanza di alimentazione. Il tempo necessario per accedere alla memoria di massa risulta però di gran lunga superiore rispetto a quello necessario per accedere alla memoria centrale. L’informazione contenuta nella memoria di massa è organizzata in *archivi* o *file*.

- *Periferiche.*

Sono dispositivi che permettono la comunicazione tra il computer e l'ambiente esterno. Esempi di periferiche sono la tastiera, il video, il mouse, il modem, la stampante, etc.

- *Bus.*

È essenzialmente un insieme di cavi, impiegati per collegare tra loro i componenti indicati sopra.

Software

I programmi utilizzabili su un calcolatore possono svolgere diversi compiti. È possibile suddividere l'insieme dei programmi nei seguenti tre gruppi fondamentali.

- *Sistema operativo.*

Ha il compito di controllare e coordinare l'uso di tutte le risorse della macchina. Alcune funzionalità essenziali svolte dal sistema operativo sono:

- ricevere ed eseguire i comandi impartiti dall'utente tramite un'opportuna interfaccia detta *shell*
- avviare l'esecuzione dei programmi
- rendere disponibili le risorse (memoria, rete, periferiche, etc.)
- permettere la condivisione ottimale delle risorse tra più utenti.

Il sistema operativo si occupa, tra l'altro, della gestione e dell'accesso al *file system*, cioè all'insieme di tutti i file presenti nella memoria di massa.

- *Utility e software di base.*

Sono programmi che consentono attività legate alla gestione della macchina e dei file (ad esempio programmi per la copia di file, programmi compressori, antivirus, etc.) o strumenti di sviluppo (editor, compilatori, interpreti, linker, etc.).

- *Programmi applicativi.*

Sono programmi che svolgono direttamente le funzionalità che interessano all'utente finale.

1.5 La macchina di von Neumann

Abbiamo visto quali sono i componenti fondamentali dell'hardware di un elaboratore. In particolare, il "cuore" dell'elaboratore è costituito da due componenti: la *memoria*, che contiene il programma da eseguire e i dati da esso utilizzati, e il *processore*, cioè l'esecutore. Questo modello, proposto da John von Neumann nel 1946, è impiegato per tutti gli elaboratori convenzionali.

Il processore opera ripetendo ciclicamente le seguenti operazioni.

- preleva dalla memoria la prossima istruzione da eseguire (fase di *Fetch*);
- interpreta l'istruzione, cioè ne riconosce il significato (fase di *Decode*);
- esegue le operazioni corrispondenti all'istruzione (fase di *Execute*).

Possiamo subito osservare che le istruzioni da eseguire, e dunque i programmi, devono essere scritti in un linguaggio comprensibile dal processore, detto *linguaggio macchina*. Inoltre il processore non è in grado di elaborare direttamente i dati contenuti nella memoria centrale, ma può operare solo su dati che si trovano all'interno di appositi registri contenuti nel processore stesso. Pertanto, per effettuare un'operazione su dati contenuti nella memoria, è necessario trasferire i dati dalla memoria nei registri del processore, effettuare l'operazione utilizzando i registri e trasferire il risultato nella memoria.

Per chiarire meglio questi aspetti presentiamo alcuni esempi. Supponiamo di disporre di un processore dotato di due registri, R1 e R2. Vogliamo calcolare la somma dei valori contenuti in due locazioni di memoria *a* e *b*, e porre il risultato in una locazione *c*.

Le operazioni da compiere saranno le seguenti.

1. Copia il contenuto della cella di memoria *a* nel registro R1.
2. Copia il contenuto della cella di memoria *b* nel registro R2.
3. Somma il contenuto dei registri R1 e R2, e poni il risultato in R1.
4. Copia il contenuto del registro R1 nella cella di memoria *c*.

Le operazioni precedenti corrispondono a istruzioni elementari del processore; in particolare:

- *Istruzioni di trasferimento dati tra memoria e processore.*

L'istruzione che permette di "caricare" in un registro R il valore contenuto in una cella di memoria di indirizzo *x* sarà indicata con LOAD R, *x*.

L'istruzione che permette di "scaricare" o, meglio, copiare nella cella di indirizzo *x* il valore contenuto nel registro R sarà indicata con STORE R, *x*.

- *Istruzioni aritmetico-logiche.*

Permettono di effettuare operazioni aritmetiche o logiche tra i dati contenuti nei registri, e lasciano il risultato in un registro. Ad esempio l'istruzione ADD R', R'' calcola la somma dei valori contenuti in R' e R'', ponendo il risultato nel primo registro specificato, cioè R'. Analogamente, nei nostri esempi considereremo istruzioni per il calcolo della sottrazione, del prodotto e della divisione intera, indicate rispettivamente con SUB, MUL e DIV.

- *Istruzioni di controllo e di salto.*

Permettono di cambiare l'ordine con cui vengono eseguite le istruzioni, e saranno descritte successivamente.

I nomi come LOAD, STORE, ADD, etc., che abbiamo utilizzato per indicare le istruzioni, sono codici mnemonici che ci aiutano a ricordare le operazioni svolte (cioè la *semantica* delle istruzioni).

In realtà nel linguaggio macchina, cioè nel linguaggio del processore, a ognuno di questi nomi corrisponde una sequenza di cifre binarie. Pertanto, per scrivere un programma direttamente in linguaggio macchina occorre utilizzare, in luogo dei codici mnemonici, le cifre binarie corrispondenti. Ricordiamo, inoltre, che questo è solo un esempio. Ogni processore ha un proprio formato di istruzioni macchina. Tuttavia le classi di istruzioni presenti, sebbene molto più articolate, sono analoghe per la maggior parte dei processori.

Il linguaggio costituito dai codici mnemonici associati alle istruzioni macchina è detto *linguaggio assembler*. Chiaramente, ogni processore ha un proprio linguaggio assembler.

Utilizzando i codici mnemonici appena introdotti, l'assegnamento alla locazione *c* della somma dei contenuti delle locazioni di memoria *a* e *b* può essere effettuato mediante le seguenti istruzioni:

```
LOAD R1, a
LOAD R2, b
ADD R1,R2
STORE R1, c
```

(In realtà, in luogo di *a*, *b* e *c* andranno scritti gli indirizzi di memoria corrispondenti).

Le istruzioni di controllo *c* di salto permettono di modificare l'ordine, strettamente sequenziale, con cui vengono eseguite le istruzioni di un programma. È possibile associare un'etichetta a un'istruzione scrivendo, ad esempio:

```
et: LOAD R1, a
```

L'istruzione JUMP *et* fa proseguire l'esecuzione dall'istruzione preceduta dall'etichetta *et*. L'istruzione JZERO *R*, *et* permette invece di decidere quale sarà la prossima istruzione da eseguire in base al valore contenuto nel registro *R*: in particolare, se *R* contiene zero, l'esecuzione prosegue dall'istruzione preceduta dall'etichetta *et*; se invece *R* contiene un valore diverso da zero, l'esecuzione prosegue normalmente con l'istruzione successiva.

Ad esempio l'esecuzione del codice:

```
LOAD R1, 30
LOAD R2, 31
SUB R1, R2
JZERO R1, alfa
LOAD R1, 30
ADD R1, R2
alfa: STORE R1, 40
```

ha l'effetto di porre nella cella 40 la somma dei valori contenuti nelle celle 30 e 31, nel caso essi siano diversi, e di porre nella cella 40 il valore zero, nel caso essi siano uguali.

Presentiamo ora un esempio più articolato: il calcolo del massimo comun divisore tra due numeri mediante l'algoritmo di Euclide che, ricordiamo, è il seguente:

1. Siano x e y i due numeri.
2. Calcola il resto della divisione di x per y .
3. Se il resto è diverso da zero, ricomincia dal passo 2
utilizzando come x il valore attuale di y , e come y il valore del resto,
altrimenti prosegui con il passo successivo.
4. Il massimo comun divisore è uguale al valore attuale di y .

Supponiamo che i due numeri si trovino agli indirizzi di memoria 101 e 102, e che il risultato vada posto all'indirizzo 103. Per semplicità supponiamo inoltre di poter cambiare i contenuti delle locazioni 101 e 102 durante il calcolo (questo comporta la perdita dei dati iniziali).

All'inizio dell'algoritmo di Euclide, i numeri di cui si vuole calcolare il massimo comun divisore vengono indicati con x e y . Pertanto x corrisponderà alla locazione 101 e y alla locazione 102. Per effettuare le operazioni è necessario trasferire i numeri nei registri; pertanto, le prime istruzioni saranno:

```
LOAD R1, 101
LOAD R2, 102
```

Il passo successivo è il calcolo del resto della divisione di x per y . Si osservi che, indicando con il simbolo $/$ la divisione intera, il resto è dato dal risultato dell'espressione $x - (x/y) * y$.

Per il calcolo di tale espressione possiamo eseguire i passi sottoelencati.

- Calcoliamo x/y con l'istruzione DIV R1, R2. Questa istruzione lascia il risultato in R1 (che dunque non conterrà più il valore di x).
- Calcoliamo $(x/y) * y$ moltiplicando il risultato precedentemente ottenuto per il valore di y con l'istruzione MUL R1, R2. A questo punto R1 contiene il risultato dell'espressione $(x/y) * y$, che andrà sottratto da x .
- Calcoliamo $x - (x/y) * y$. Per farlo, carichiamo in R2 il valore di x utilizzando l'istruzione LOAD R2, 101, e sottraiamo da R2 il contenuto di R1 utilizzando l'istruzione SUB R2, R1. Si osservi che, poiché il risultato viene lasciato nel primo registro specificato, esso si troverà in R2.

A questo punto R2 contiene il resto della divisione. Possiamo quindi espandere il Passo 3 dell'algoritmo di Euclide. Se il registro R2 non contiene zero, dopo avere modificato i valori di x e y occorre ricominciare dal Passo 2; se il registro R2 contiene zero si prosegue invece con il Passo 4. Utilizziamo l'istruzione di *salto condizionato* JZERO R2, fine per trattare il caso di resto zero. Dovremo poi scrivere un blocco di istruzioni, che inizi con l'etichetta fine, per rappresentare il Passo 4 dell'algoritmo.

Quando il resto è diverso da zero, prima di ripetere dal Passo 2 occorre spostare il valore attuale di y in x (operazione che può essere realizzata con le due istruzioni LOAD R1, 102 e STORE R1, 101) e copiare in y il valore del resto, che si trova in R2, mediante l'istruzione

STORE R2, 102. Si noti che a questo punto R1 e R2 contengono i nuovi valori di x e y , e si può quindi riprendere dall'istruzione DIV R1, R2 scritta in precedenza. Per fare questo, introduciamo l'istruzione di *salto non condizionato* JUMP alfa, dove l'etichetta alfa sarà associata all'istruzione di divisione.

Infine dobbiamo scrivere le istruzioni corrispondenti al Passo 4, il cui compito è quello di lasciare il massimo comun divisore, che secondo l'algoritmo di Euclide si trova in y (cioè nella cella 102), nella cella 103. Alla prima di queste istruzioni dev'essere associata l'etichetta fine. Il codice è dunque:

```
fine: LOAD R1, 102
      STORE R1, 103
```

Ecco il codice completo:

```
LOAD R1, 101
LOAD R2, 102
alfa: DIV R1, R2
      MUL R1, R2
      LOAD R2, 101
      SUB R2, R1
      JZERO R2, fine
      LOAD R1, 102
      STORE R1, 101
      STORE R2, 102
      JUMP alfa
fine: LOAD R1, 102
      STORE R1, 103
```

Per comprendere meglio il comportamento del codice presentato sopra, si consiglia di utilizzare carta e penna per simularne l'esecuzione da parte della macchina.

1.6 Dal linguaggio macchina ai linguaggi ad alto livello

Gli esempi presentati mostrano che, usando il linguaggio macchina, anche il calcolo di semplici espressioni aritmetiche dev'essere trasformato in una lunga e poco comprensibile sequenza di istruzioni. La programmazione in linguaggio macchina, praticata fino agli inizi degli anni Cinquanta, presenta pertanto notevoli svantaggi.

- È necessario conoscere i dettagli dell'architettura del processore utilizzato e il relativo linguaggio (che è formato da una sequenza poco leggibile di codici).
- Poiché ogni processore ha un proprio linguaggio macchina, risulta impossibile trasferire i programmi da una macchina a un'altra.

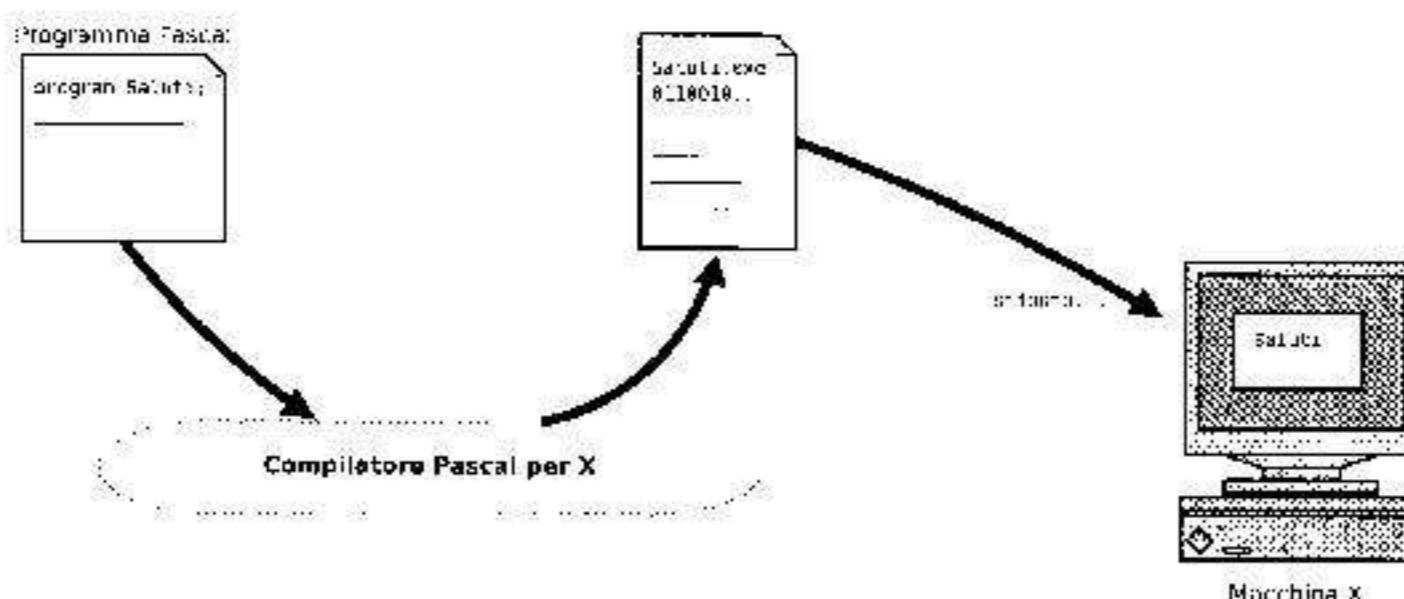


Figura 1.1 Dal sorgente Pascal all'eseguibile.

- Il programmatore deve conoscere in dettaglio tutte le caratteristiche della macchina che utilizza.
- Il programmatore si specializza nell'uso di "trucchi" legati alle caratteristiche specifiche della macchina. In questo modo i programmi risultano pressoché incomprensibili e difficilmente modificabili perfino dalla stessa persona che li ha scritti.
- Leggendo il programma è difficile comprenderne la struttura e individuare eventuali errori.
- È necessario riscrivere completamente i programmi quando vengono trasferiti su una macchina differente.

In sintesi possiamo osservare che tutta l'attività di programmazione in linguaggio macchina è strettamente dipendente dalle caratteristiche del processore utilizzato, alle quali il programmatore è costretto ad adeguarsi.

Per svincolare la programmazione dalle caratteristiche peculiari della macchina utilizzata, sono stati introdotti i cosiddetti *linguaggi ad alto livello*. Essi non sono pensati per essere compresi direttamente da macchine reali, ma da macchine "astratte", in grado di effettuare operazioni di più alto livello, rispetto alle operazioni elementari dei processori reali. In questo modo l'attività di programmazione viene affrancata dalla conoscenza dei dettagli architettonici della macchina utilizzata.

Chiaramente, per poter poi eseguire su una macchina reale un programma scritto in linguaggio ad alto livello, è necessario trasformarlo, o meglio *tradurlo*, nel linguaggio della macchina utilizzata. Questa operazione, fortunatamente, può essere effettuata in modo automatico, cioè usando appositi programmi che prendono il nome di *compilatori*. Ad esempio, come schematizzato nella Figura 1.1, un compilatore Pascal per una macchina *X* è un programma che riceve in ingresso un programma *P* scritto in linguaggio Pascal, e produce in uscita un programma *P'* scritto nel linguaggio della macchina *X* equivalente a *P*, cioè tale che le esecuzioni di *P* e di *P'* sugli stessi dati di ingresso producono gli stessi risultati.

Un programmatore che conosca il linguaggio Pascal e che disponga di un compilatore Pascal per la macchina *X* potrà scrivere programmi da eseguire su *X* senza conoscerne il linguaggio

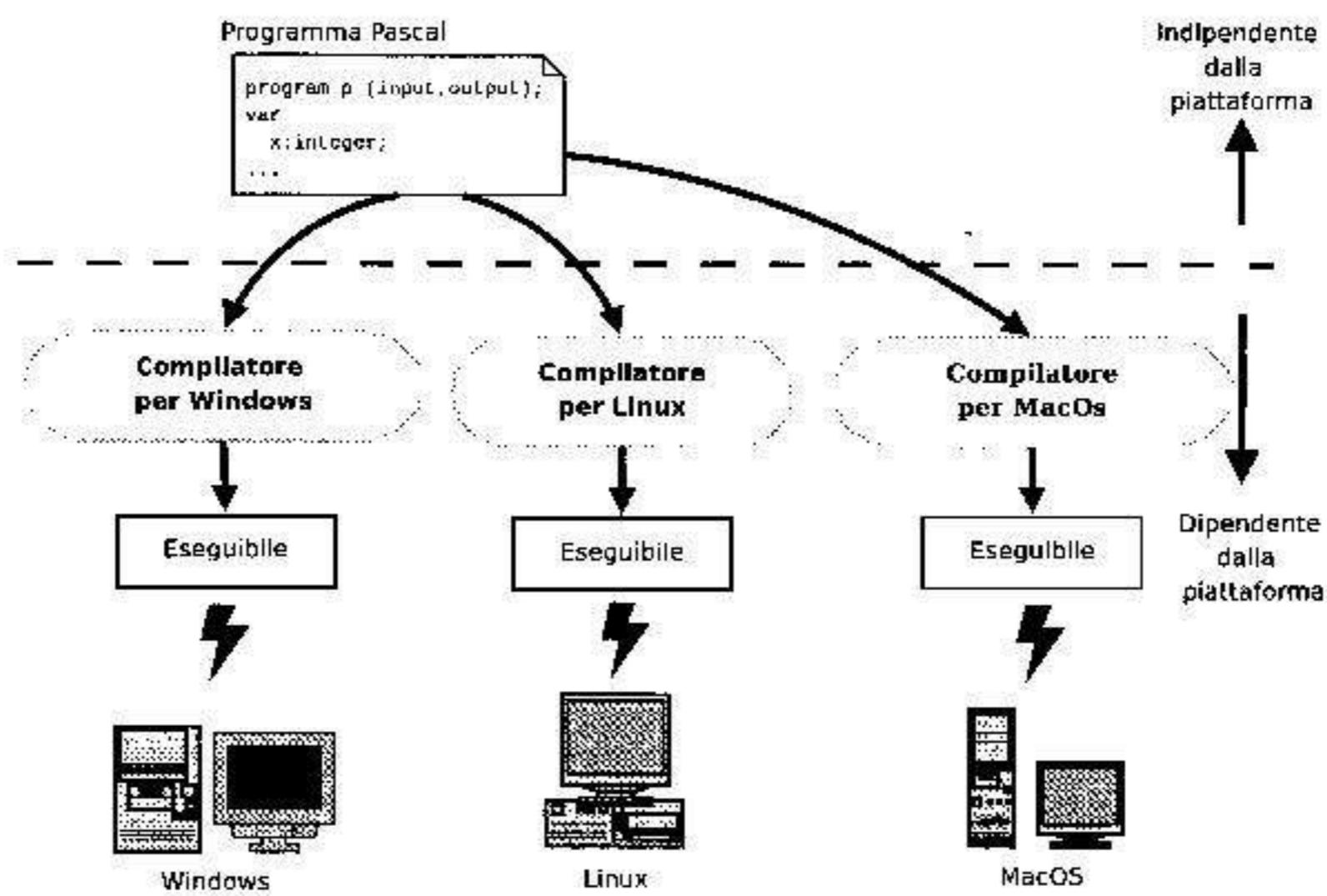


Figura 1.2 Indipendenza del programma sorgente dalla piattaforma.

macchina. Inoltre, disponendo di un compilatore Pascal per un'altra macchina Y , gli stessi programmi potranno essere eseguiti da Y senza doverli riscrivere da zero, al contrario di quanto succedeva programmando in linguaggio macchina. In pratica, il programmatore scrive il proprio programma facendo riferimento a una macchina astratta, la “macchina Pascal”; grazie al compilatore, un programma per la “macchina Pascal” potrà essere trasformato in un programma per una macchina reale. La conoscenza del linguaggio macchina è necessaria solo per costruire il compilatore (o meglio solo per costruire una piccola parte del compilatore, quella che genera il codice finale).

In questo modo, l'introduzione del livello di astrazione fornito dal compilatore rende i programmi ad alto livello portabili su macchine diverse, come illustrato nella Figura 1.2.

Gli *interpreti* sono una variante dei compilatori. Un interprete è un programma che simula direttamente una macchina astratta. Anziché effettuare la traduzione di P , un interprete Pascal legge ogni istruzione contenuta nel programma P ed effettua immediatamente, tramite la macchina X , le operazioni corrispondenti all'istruzione letta. In pratica, la traduzione dell'intero programma prima dell'esecuzione viene sostituita dalla traduzione simultanea, con esecuzione immediata di ciascuna istruzione. Si noti che, dovendo eseguire il programma più volte, risulta più vantaggioso, in termini di tempo, impiegare un compilatore, in quanto la traduzione viene effettuata una volta per tutte.

Segnaliamo infine l'esistenza degli *assemblatori*, che si occupano di tradurre in linguaggio macchina i programmi scritti con codici mnemonici corrispondenti alle istruzioni macchina, cioè programmi scritti in un linguaggio *assembler*.

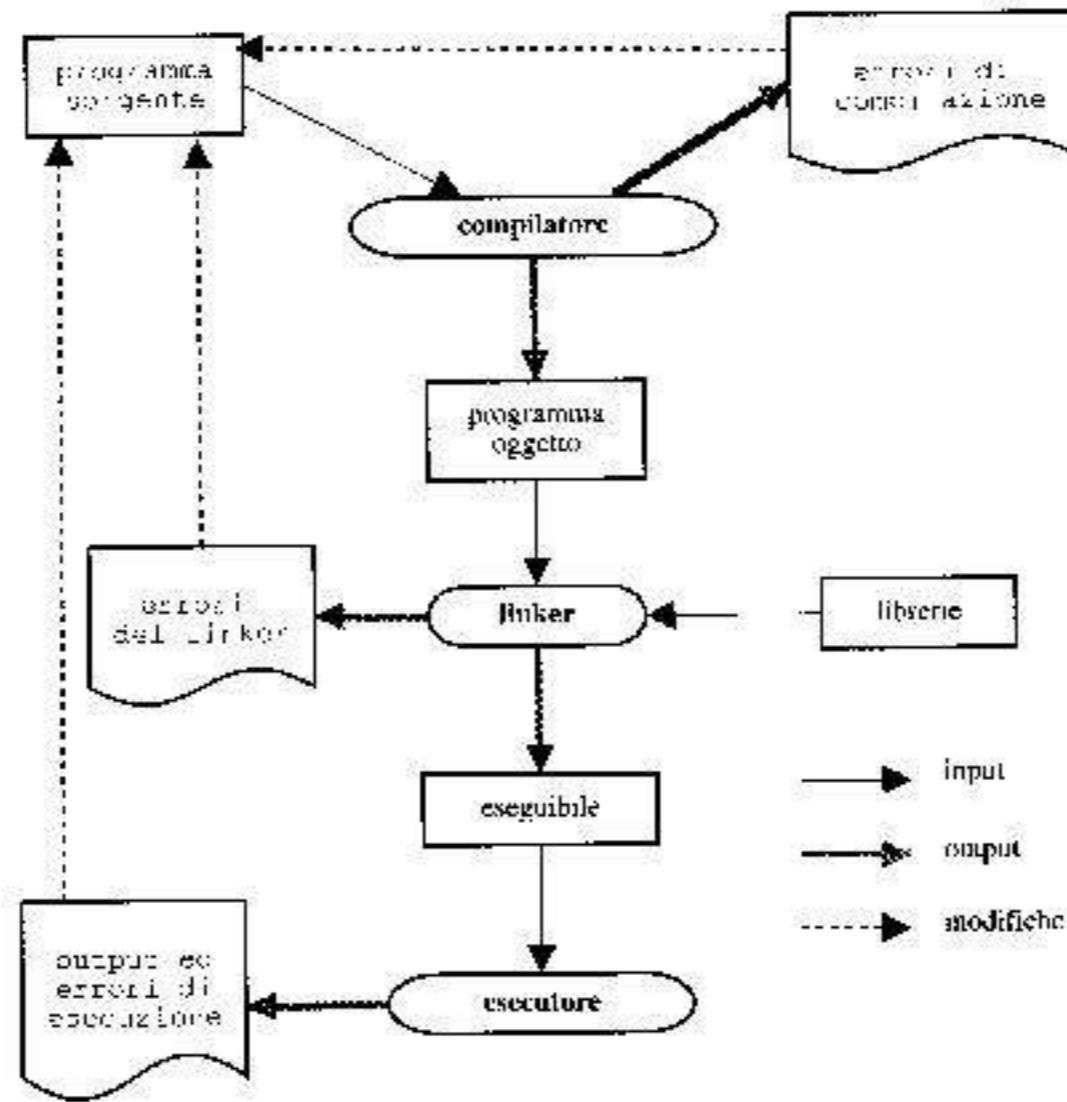


Figura 1.3 Ciclo di vita di un programma.

1.7 Strumenti per la stesura dei programmi

Nella fase di preparazione dei programmi si usano vari strumenti automatici (cioè altri programmi). Il primo di tutti è l'*editor*, un programma che permette di scrivere testi. Utilizzando un editor è possibile scrivere in un file il testo del programma nel linguaggio ad alto livello. Tale testo viene chiamato *programma sorgente* o *codice sorgente*.

Il programma sorgente può essere quindi dato in ingresso al compilatore che lo traduce nel *codice oggetto*, scritto in linguaggio macchina.

Viene poi utilizzato uno strumento, detto *linker* (letteralmente “collegatore”), che ha il compito di collegare tra loro i vari moduli che costituiscono lo stesso programma. Infatti, nella maggior parte dei linguaggi è possibile suddividere il programma sorgente in più file, che vengono compilati separatamente creando diversi file oggetto. Inoltre un programma può utilizzare alcune funzioni dette di *libreria*, messe a disposizione del programmatore che le può richiamare direttamente dai programmi (ad esempio le procedure di lettura e scrittura o alcune funzioni matematiche). Il linker collega quindi tra loro i file contenenti il codice oggetto dei moduli che costituiscono un programma, unendovi anche il codice delle funzioni di libreria utilizzate e producendo un file contenente il *codice eseguibile* che corrisponde al programma di partenza. Il codice eseguibile può essere a questo punto caricato (da parte del *loader* del sistema operativo) per l'esecuzione.

Durante le fasi di preparazione di un programma possono verificarsi vari tipi di errore, che abbiamo riassunto nella Figura 1.3. Nella compilazione si possono riscontrare ad esempio errori

di *sintassi* o di *semantica* dovuti all'uso scorretto del linguaggio. In questo caso, prima di passare alle fasi successive, occorre correggere il programma sorgente utilizzando di nuovo l'editor.

Una volta che il programma sorgente viene compilato senza errori, si può richiamare il linker. Anche qui possono verificarsi diversi tipi di errore, quali la mancanza di uno o più moduli del programma oppure l'utilizzo scorretto di funzioni di libreria. Se ci sono errori, è necessario richiamare nuovamente il linker, in taluni casi dopo aver anche corretto tramite l'editor, e compilato di nuovo, uno o più moduli del programma sorgente.

Una volta che il linker non ha riscontrato errori, è possibile mandare in esecuzione il codice oggetto. Anche durante l'esecuzione possono verificarsi errori che la interrompono. Ad esempio, se si incontra un'operazione di divisione, e il divisore vale zero, l'esecuzione è interrotta. Si noti che gli errori di esecuzione possono dipendere dai dati in ingresso: per alcuni dati, lo stesso programma può terminare correttamente la propria esecuzione, mentre per altri potrebbe interrompersi, riscontrando un errore.

L'assenza di errori in esecuzione non implica che il programma sia corretto. Infatti il programma potrebbe produrre risultati diversi da quelli attesi, cioè svolgere una funzione diversa da quella per cui è stato creato, oppure potrebbe non produrre alcun risultato e continuare la propria esecuzione all'infinito. La fase di testing, che ha come obiettivo quello di verificare che il programma sia corretto rispetto alle specifiche in base alle quali è stato costruito, è una delle fasi più difficili e delicate. Molte volte i comportamenti anomali si verificano solo per particolari valori di ingresso. Inoltre, già in programmi di poche decine di righe, può essere difficile risalire alla causa di un errore. Questa fase è detta di *debugging*. Uno strumento utile in questa fase è il *debugger*, che permette di osservare passo dopo passo l'andamento dell'esecuzione di un programma.

Una volta che un programma è corretto e funzionante, può ancora essere soggetto a modifiche, ad esempio per aggiungere nuove funzionalità oltre a quelle per cui era stato inizialmente progettato. Questa fase prende il nome di *manutenzione* e spesso è notevolmente complessa. Per ridurre i costi di manutenzione bisogna curare particolarmente la stesura iniziale dei programmi in modo che risultino ben documentati e leggibili. Questo faciliterà la comprensione successiva del significato e del funzionamento del codice.

Per i personal computer sono disponibili ambienti per lo sviluppo dei programmi (IDE, *Integrated Development Environment*) in cui gli strumenti sopra menzionati: editor, compilatore, linker e, talvolta, debugger, sono tra loro integrati. In questo caso, una volta terminata la scrittura del programma con l'editor, è possibile mediante un solo comando richiedere la compilazione, il "linkaggio" e l'esecuzione del programma stesso.

1.8 Java Virtual Machine

Nel caso del *linguaggio Java*, per la traduzione dei programmi viene utilizzata una soluzione intermedia, che sfrutta sia la compilazione sia l'interpretazione. I progettisti del linguaggio Java hanno infatti definito, oltre al linguaggio stesso, la *Java Virtual Machine*¹ (JVM), una sor-

¹ Macchina virtuale Java.

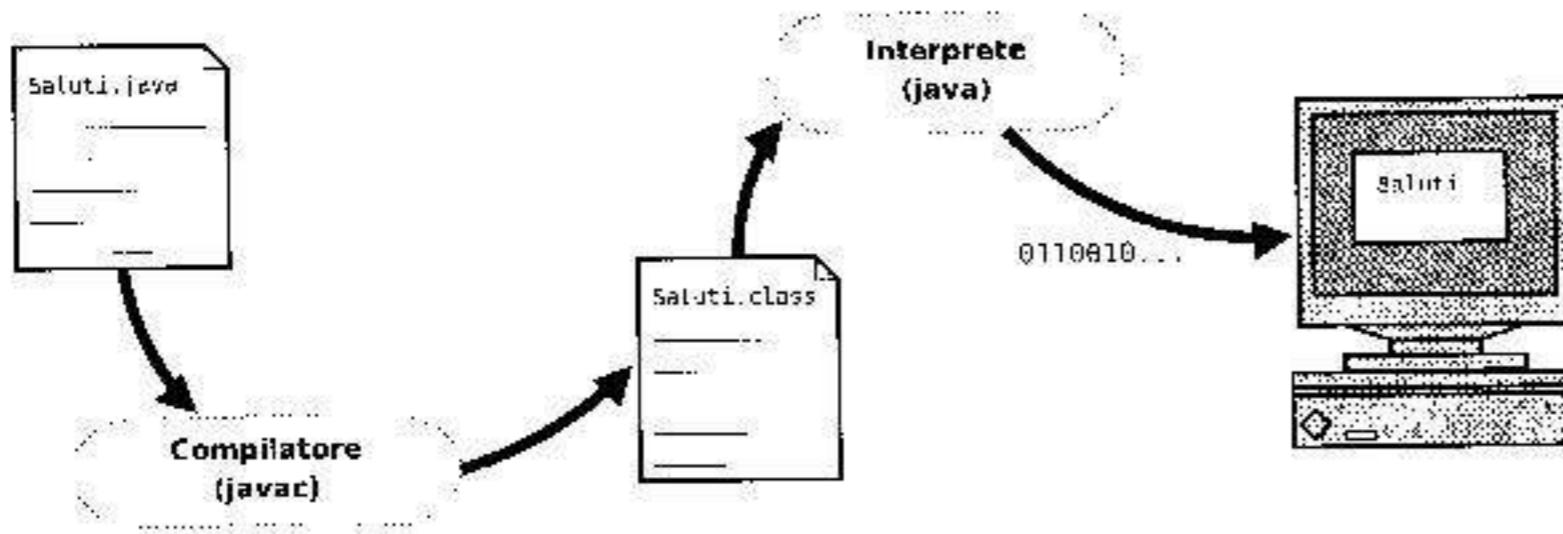


Figura 1.4 Creazione di un'applicazione Java.

ta di macchina astratta in grado di eseguire un codice a basso livello (cioè simile a un codice macchina), che prende il nome di *bytecode*.

Come evidenziato nella Figura 1.4, il compilatore Java traduce il programma sorgente (nell'esempio contenuto nel file `Saluti.java`) in un programma equivalente in bytecode (memorizzato nel file `Saluti.class`) per la Java Virtual Machine. Per eseguire il programma occorre utilizzare un interprete del bytecode che simuli la Java Virtual Machine sulla macchina reale impiegata.

In questa soluzione, la fase di compilazione è indipendente dalla macchina reale di cui si dispone: il bytecode è infatti prodotto per la macchina virtuale. L'interprete del bytecode dipenderà invece dalla specifica macchina utilizzata per l'esecuzione. In questo modo si ottiene, oltre alla portabilità del programma sorgente, anche quella del compilato (il bytecode), come illustrato nella Figura 1.5.

Al pari di tutti i linguaggi, anche i programmi Java utilizzano codice di libreria. In particolare, il linguaggio Java dispone di una libreria molto ampia. Il codice contenuto nelle librerie (che in Java vengono chiamate *package*) è già compilato sotto forma di bytecode.

Durante la compilazione di un programma il compilatore accede alle librerie per verificare l'uso corretto delle risorse disponibili in esse (vedremo più avanti che in un programma occorre indicare quali librerie si utilizzano). Tuttavia il codice prodotto dal compilatore contiene solo il bytecode corrispondente al programma che si è tradotto, e non il codice di libreria richiamato dal programma. Per eseguire il programma, la macchina virtuale carica, oltre al bytecode del programma da eseguire, i bytecode delle parti di librerie impiegate dal programma.

I browser web odierni contengono un interprete di bytecode, ossia una Java Virtual Machine. In questo modo, il browser è in grado di eseguire applicazioni Java contenute sotto forma di bytecode in pagine web remote. Queste applicazioni sono dette *applet*.

Quando si raggiunge una pagina contenente un'applet, questa viene trasferita, tramite la rete, al browser, che è in grado di esegirla utilizzando la propria Java Virtual Machine. Si noti che il codice dell'applet, essendo bytecode, è indipendente dalla macchina reale su cui avverrà l'esecuzione. Grazie a ciò, chi scrive un'applet non ha bisogno di conoscere la macchina su cui questa sarà eseguita. Più in generale, l'intero linguaggio Java è stato pensato per essere indipendente dalla piattaforma.

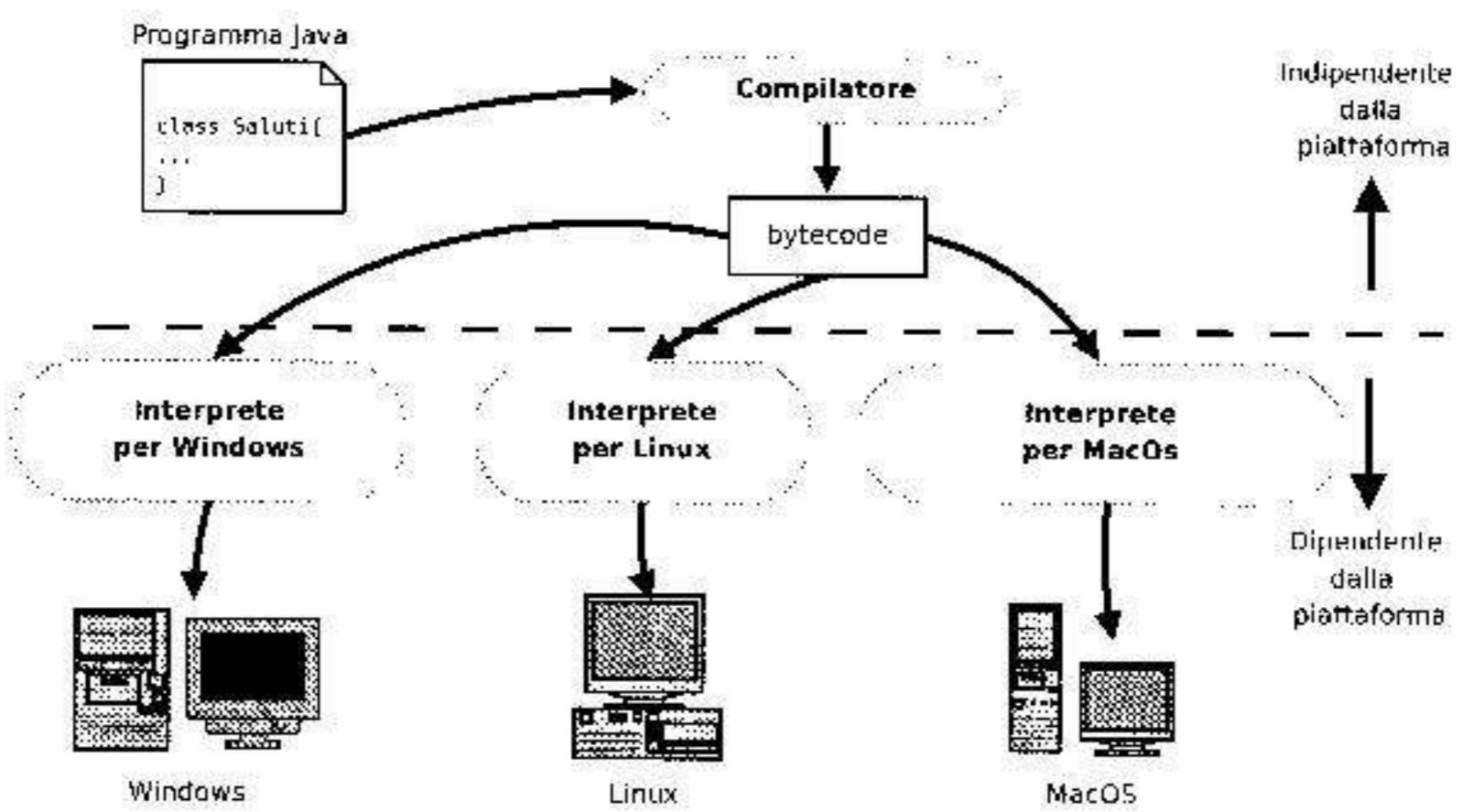


Figura 1.5 Indipendenza del bytecode dalla piattaforma.

1.9 Programmazione strutturata

Abbiamo visto che un programma è costituito da istruzioni elementari. Per descrivere algoritmi significativi dev'essere possibile associare a un programma più sequenze d'esecuzione che dipendano da situazioni che si creano durante l'esecuzione. Ad esempio un programma per il calcolo della divisione tra due numeri dovrà effettuare la divisione solo nel caso in cui il divisore sia diverso da zero, e dovrà stampare un messaggio d'errore in caso contrario. Nella *programmazione strutturata*, proposta come metodologia per la stesura di programmi agli inizi degli anni Settanta, l'esecutore è guidato alla sequenza di esecuzione opportuna, tra tutte quelle possibili, mediante tre strutture di controllo fondamentali:

- la *sequenza* che permette di eseguire le istruzioni secondo l'ordine in cui sono scritte;
 - la *selezione* che permette di scegliere l'esecuzione di un blocco di istruzioni tra due possibili in base al valore di una condizione;
 - l'*iterazione* che permette di ripetere l'esecuzione di un blocco di istruzioni un certo numero di volte.

È stato dimostrato che i programmi esprimibili tramite istruzioni di salto (*goto*) o diagrammi di flusso (*flow-chart*) possono essere riscritti utilizzando esclusivamente le tre strutture di controllo fondamentali. Inoltre l'impiego di queste strutture migliora la qualità dei programmi prodotti, la cui architettura risulta maggiormente comprensibile, riducendo al contempo la possibilità di errori. La programmazione strutturata si è imposta come metodologia nella stesura dei programmi: quasi tutti i linguaggi di programmazione (compresi Pascal, C e Java) hanno costrutti che permettono di programmare utilizzando direttamente le strutture di controllo fondamentali.

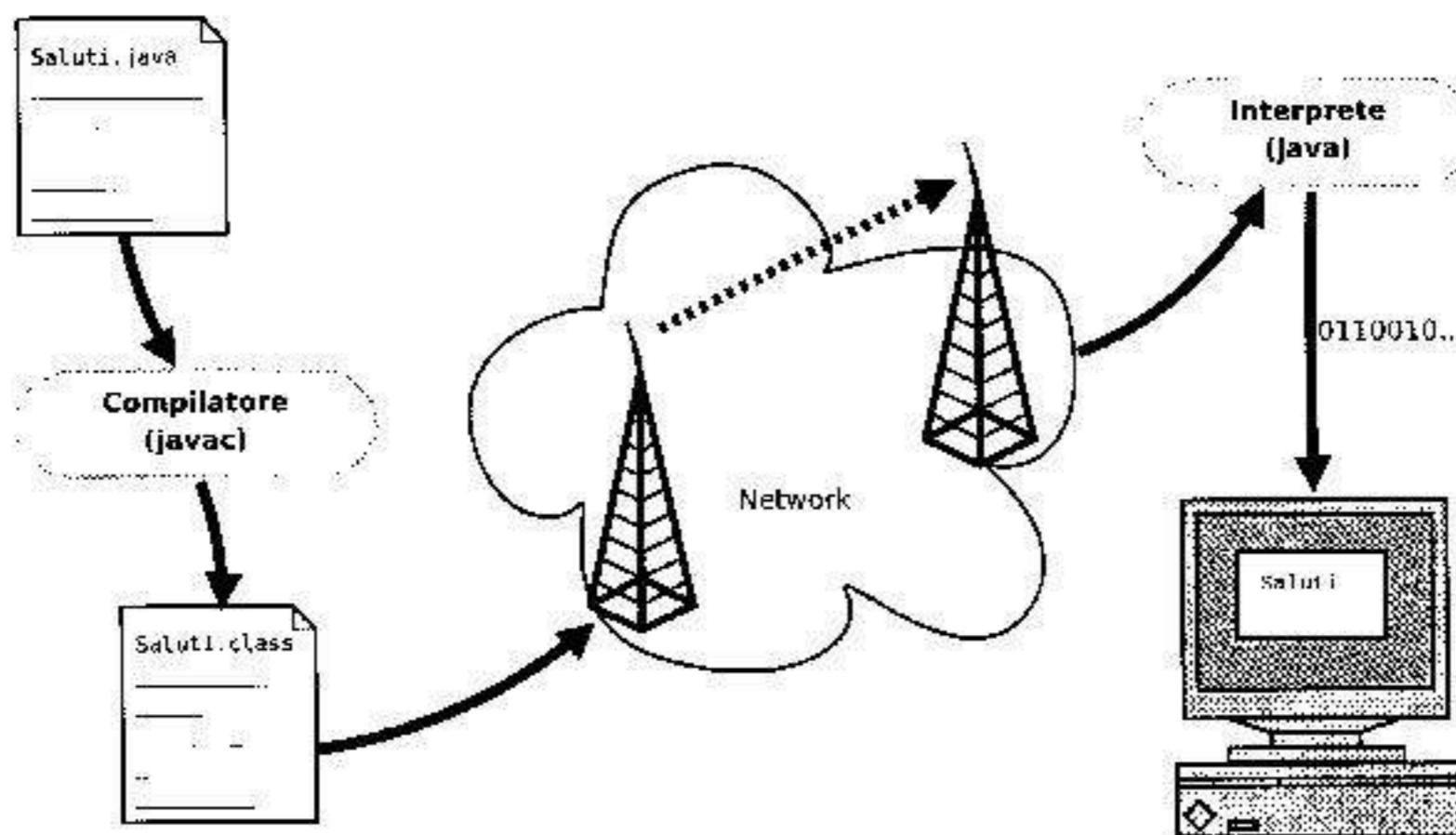


Figura 1.6 Esecuzione remota.

Sequenza

In mancanza di indicazioni diverse, le istruzioni sono eseguite nello stesso ordine in cui compaiono nel programma, cioè secondo la *sequenza* in cui sono scritte. Ad esempio la somma di due numeri può essere calcolata, utilizzando esclusivamente la sequenza, mediante le seguenti operazioni:

leggi i numeri a, b
calcola $a + b$
scrivi il risultato

Selezione

La *selezione* permette di scegliere un'istruzione o un blocco di istruzioni da eseguire tra due alternative. In ogni caso, terminata l'esecuzione del blocco selezionato, si prosegue dall'istruzione successiva al costrutto di selezione.

Possiamo schematizzare la selezione così:

```
SE condizione
  ALLORA
    blocco1
  ALTRIMENTI
    blocco2
  FINESE
```

L'esecuzione avviene come segue. Prima di tutto è valutata la *condizione*: se questa risulta vera, vengono eseguite le istruzioni del *blocco1*; se risulta falsa, vengono eseguite quelle del *blocco2*.

In ambedue i casi, l'esecuzione procede con l'istruzione che segue immediatamente la fine del costrutto di selezione, cioè con l'istruzione che segue la parola FINESE.

Nel caso in cui il *blocco2* non contenga alcuna istruzione, si può utilizzare il seguente schema semplificato:

```
SE condizione
  ALLORA
    blocco1
  FINESE
```

Ad esempio, il calcolo della divisione tra due numeri può essere effettuato controllando che il divisore sia diverso da zero, in questo modo:

```
leggi il dividendo e il divisore
SE il divisore è diverso da zero
  ALLORA
    calcola dividendo/divisore
    scrivi il risultato
  ALTRIMENTI
    scrivi "errore: divisione per zero"
  FINESE
```

Notiamo che, in base al valore del divisore, è possibile selezionare una tra due possibili sequenze di esecuzione. Dopo l'esecuzione del ramo scelto, l'elaborazione riprende da ciò che segue la parola FINESE (in questo caso, non essendoci nulla, l'esecuzione termina).

Calcolo delle soluzioni di un'equazione di secondo grado

Utilizzando le strutture fondamentali di sequenza e selezione descriviamo un algoritmo per il calcolo delle radici di un'equazione di secondo grado della forma $ax^2 + bx + c = 0$.

Ricordiamo che il numero di soluzioni dipende dal valore della quantità $b^2 - 4ac$, detta anche *discriminante*. Si possono verificare le seguenti situazioni:

- il discriminante è negativo: l'equazione non ammette soluzioni reali;
- il discriminante è nullo: l'equazione ammette due soluzioni reali coincidenti, di valore $\frac{-b}{2a}$;
- il discriminante è positivo: l'equazione ammette due soluzioni reali distinte, date dalla formula $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

Il procedimento risolutivo dell'equazione può essere dunque descritto dal seguente algoritmo:

```
leggi i valori dei parametri a, b, c
calcola il discriminante
SE il discriminante è minore di zero
  ALLORA
    scrivi "nessuna soluzione reale"
```

ALTRIMENTI

SE il discriminante è uguale a zero

ALLORAcalcola $\frac{-b}{2a}$

scrivi "Due soluzioni coincidenti: ", il risultato

ALTRIMENTI

scrivi "Due soluzioni: ",

calcola $\frac{-b - \sqrt{b^2 - 4ac}}{2a}$

scrivi il risultato

calcola $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$

scrivi il risultato

FINESE**FINESE**

Si osservi che nell'algoritmo appena descritto possono verificarsi tre differenti sequenze d'esecuzione. Nell'algoritmo sono utilizzate istruzioni di tre tipi:

- di *lettura*, per leggere dall'ambiente esterno i valori da elaborare;
- di *scrittura*, per comunicare all'ambiente esterno i risultati;
- di *calcolo*, per calcolare il valore di espressioni aritmetiche.

Iterazione

L'*iterazione* permette di ripetere l'esecuzione di una o più istruzioni in base al valore di una condizione.

Lo schema dell'iterazione è questo:

ESEGUI*blocco***QUANDO** *condizione*

Anzitutto viene eseguito il *blocco* di istruzioni. Quindi si valuta la *condizione*. Se questa risulta vera, si ripete eseguendo nuovamente il *blocco* e valutando ancora la *condizione*; se la *condizione* risulta falsa, si prosegue con l'istruzione scritta dopo il costrutto iterativo. Si noti in particolare che:

- il *blocco* è eseguito sempre *almeno una volta*, in quanto la *condizione* viene valutata in coda;
- l'esecuzione del costrutto iterativo termina quando la *condizione* diventa falsa.

Presentiamo un altro schema iterativo.

```
QUANDO condizione ESEGUI
  blocco
RIPETI
```

In questo caso viene valutata prima di tutto la *condizione*. Se essa risulta vera, allora si esegue il *blocco* e si valuta nuovamente la *condizione*. Se la *condizione* risulta falsa, si passa a eseguire l'istruzione che segue il costrutto iterativo, cioè l'istruzione che segue la parola RIPETI. Pertanto:

- il *blocco* può essere eseguito anche zero volte, in quanto la *condizione* viene valutata in testa;
- l'esecuzione del costrutto iterativo termina quando la *condizione* diventa falsa.

Osserviamo che il comportamento dello schema QUANDO...RIPETI può essere simulato combinando lo schema della selezione con lo schema ESEGUI...QUANDO..., come segue:

```
SE condizione
  ALLORA
    ESEGUI
      blocco
    QUANDO condizione
  FINESE
```

Calcolo della somma dei primi 100 numeri interi

Vogliamo calcolare la somma dei primi 100 numeri interi.^{*} Questo problema potrebbe essere risolto utilizzando semplicemente la formula di Gauss ($\sum_{i=1}^n i = \frac{n(n+1)}{2}$). Tuttavia, al fine di mostrare l'uso dei costrutti iterativi, scriviamo un algoritmo che calcola iterativamente tale somma.

```
poni il valore della somma a zero
inizia a considerare il numero 1
ESEGUI
  aggiungi alla somma il numero che stai considerando
  considera il numero successivo
QUANDO il numero che stai considerando non supera 100
  scrivi la somma
```

1.10 Variabili e assegnamenti

Nell'esempio relativo alla soluzione delle equazioni di secondo grado sono stati utilizzati i nomi *a*, *b* e *c* per indicare i valori dei coefficienti forniti dall'esterno. È stato inoltre utilizzato il nome discriminante per indicare il valore del discriminante calcolato subito dopo la lettura dei dati. Possiamo immaginare che queste quantità siano immagazzinate in appositi contenitori (detti *variabili*), che chiameremo proprio *a*, *b*, *c* e *discriminante*. È possibile assegnare un valore a una variabile mediante un'istruzione di lettura, come:

leggi a, b, c

in cui si ricevono dall'ambiente esterno tre valori, che vengono posti, rispettivamente, nei contenitori di nomi a, b e c, oppure mediante un'istruzione di *assegnamento*. Ad esempio l'istruzione calcola il discriminante dell'algoritmo precedente può essere rappresentata tramite il seguente assegnamento alla variabile **discriminante**:

discriminante $\leftarrow b^2 - 4 * a * c$

La *semantica operazionale* (cioè il significato dato in termini di operazioni da eseguire) di un'istruzione di assegnamento della forma

variabile \leftarrow **espressione**

è la seguente:

- (1) viene calcolato il valore dell'espressione scritta a destra del simbolo \leftarrow ;
- (2) il risultato ottenuto è assegnato alla variabile (quindi posto nel contenitore) il cui nome è scritto a sinistra del simbolo \leftarrow , eliminando l'eventuale valore presente prima.

Ad esempio, per eseguire l'istruzione

x \leftarrow **y** + **z**

si recuperano innanzitutto i valori contenuti nelle due variabili y e z, si calcola la loro somma, e si pone il risultato nella variabile x eliminando il valore precedente.

Se prima dell'assegnamento

k \leftarrow **k** + 1

la variabile k conteneva il valore 7, dopo l'esecuzione di tale assegnamento k conterrà il valore 8. Si osservi che il nome k scritto a destra del simbolo \leftarrow denota il valore contenuto nella variabile k, mentre il nome k scritto a sinistra denota la variabile cui assegnare il risultato.

Molti linguaggi, tra cui purtroppo anche Java, utilizzano per l'assegnamento il simbolo $=$, usato comunemente per indicare l'uguaglianza. Questo talvolta crea confusione. Si osservi che l'assegnamento **x** \leftarrow **y** ha un effetto ben diverso dall'assegnamento **y** \leftarrow **x**.

A ogni variabile viene associato un *tipo* che specifica la classe di valori che questa può assumere (e dunque anche l'insieme di operazioni in cui può essere coinvolta). Ad esempio una variabile x di tipo intero può assumere come valori solo numeri interi, e su di essa possono essere effettuate soltanto le operazioni consentite per i numeri interi. I valori e le operazioni possibili su una variabile y di tipo carattere saranno differenti da quelli permessi su x.

Si osservi che il concetto di *variabile* è un'*astrazione* del concetto di locazione di memoria; l'assegnamento di un valore a una variabile è un'*astrazione* dell'operazione STORE. Sebbene tutte le variabili siano rappresentate nella memoria come sequenze di bit, tali sequenze possono essere interpretate diversamente in base ai tipi. Pertanto la nozione di tipo fornisce un'*astrazione* rispetto alla rappresentazione effettiva dei dati: il programmatore può utilizzare variabili di tipi differenti senza doverne conoscere l'effettiva rappresentazione.

Molti linguaggi richiedono di *dichiarare* le variabili all'inizio del programma (ad esempio Pascal) o prima del loro utilizzo (ad esempio Java) indicandone il tipo. Oltre ad accrescere la leggibilità dei programmi, ciò facilita la realizzazione di compilatori efficienti. Pertanto nei prossimi esempi indicheremo sempre i nomi e i tipi delle variabili utilizzate.

Riscriviamo ora l'algoritmo per la soluzione delle equazioni di secondo grado in base a quanto abbiamo appena visto. In particolare introduciamo alcune variabili, esplicitiamo alcuni passi mediante assegnamenti ed esprimiamo le condizioni utilizzando gli *operatori di confronto* < e ==.²

variabili a, b, c, discriminante, x, x1, x2: numeri reali

```

leggi a, b, c
discriminante ← b2 - 4 * a * c
SE discriminante < 0
    ALLORA
        scrivi "nessuna soluzione reale"
    ALTRIMENTI
        SE discriminante == 0
            ALLORA
                x ← - b / (2 * a)
                scrivi "Due soluzioni coincidenti: ", x
            ALTRIMENTI
                x1 ← (- b - √discriminante) / (2 * a)
                x2 ← (- b + √discriminante) / (2 * a)
                scrivi "Due soluzioni: ", x1, x2
        FINESE
    FINESE

```

Rivediamo ora l'esempio relativo al calcolo della somma dei primi 100 numeri interi introducendo esplicitamente alcune variabili. Occorre una variabile in cui memorizzare via via la somma. Chiameremo questa variabile somma. Serve inoltre un'altra variabile per sapere di volta in volta quale numero stiamo considerando. Poiché questa variabile funge da contatore, in quanto ci permette di contare da 1 a 100, utilizzeremo il nome cont. Al fine di aumentare la leggibilità dei programmi, è opportuno infatti scegliere nomi che ricordino che cosa rappresentano le singole variabili. In questo caso, ambedue le variabili sono destinate a contenere numeri interi.

L'algoritmo dato in precedenza può essere facilmente riscritto come segue:

variabili somma, cont: numeri interi

```

somma ← 0
cont ← 1
ESEGUI
    somma ← somma + cont

```

² In Java, come in C, l'operatore di confronto per l'uguaglianza è indicato con i simboli == per distinguerlo dall'operatore di assegnamento per il quale si utilizza =.

```

cont ← cont + 1
QUANDO cont <= 100
scrivi somma

```

Si osservi che l'assegnamento `somma ← somma + cont` ha come effetto quello di aggiungere alla variabile `somma` il valore della variabile `cont`.

Estendiamo ora l'esempio appena presentato per calcolare la somma dei numeri da 1 a n , dove n viene fornito come input. Possiamo introdurre una nuova variabile che riceve dall'esterno il valore n , fino al quale calcolare la somma, e modificare la condizione del ciclo:

variabili `somma, cont, n: numeri interi`

```

somma ← 0
cont ← 1
leggi n
ESEGUI
    somma ← somma + cont
    cont ← cont + 1
QUANDO cont <= n
scrivi somma

```

L'algoritmo precedente opera in modo corretto quando il valore fornito in ingresso è positivo. Se invece viene inserito in ingresso zero o un numero negativo, il risultato è scorretto. Infatti, se n è minore o uguale a zero, la somma dei primi n numeri è zero, mentre l'algoritmo fornirà come risultato 1.

Questo problema è dovuto al fatto che le istruzioni `somma ← somma + cont` e `cont ← cont + 1`, che si trovano all'interno del ciclo, sono sempre eseguite almeno una volta, anche quando `cont` è inizialmente già maggiore di n . Il problema può essere risolto senza difficoltà controllando la condizione all'inizio del ciclo e non alla fine o, in altre parole, utilizzando un ciclo della forma `QUANDO...RIPETI`:

variabili `somma, cont, n: numeri interi`

```

somma ← 0
cont ← 1
leggi n
QUANDO cont <= n ESEGUI
    somma ← somma + cont
    cont ← cont + 1
RIPETI
scrivi somma

```

Esercizi

- 1.1 Abbiamo visto come sia possibile simulare un ciclo della forma `QUANDO...RIPETI` con un ciclo della forma `ESEGUI...QUANDO...`. Dimostrate che è possibile anche la simulazione

inversa utilizzando un ciclo della forma QUANDO...RIPETI in sostituzione di un ciclo della forma ESEGUI...QUANDO....

- 1.2 Descrivete mediante le strutture di controllo fondamentali un algoritmo che, leggendo un numero intero, stabilisca se esso è pari o dispari.
- 1.3 Si consideri l'algoritmo per il calcolo delle radici di un'equazione di secondo grado. Che cosa succede se il coefficiente a vale zero? Modificate l'algoritmo per gestire anche questa situazione.
- 1.4 Descrivete mediante le strutture di controllo fondamentali un algoritmo che calcoli il prodotto dei numeri da 1 a n , dove il valore n viene letto dall'esterno.
- 1.5 Descrivete mediante le strutture di controllo fondamentali un algoritmo che legga una sequenza di numeri interi e scriva quanti di questi sono pari e quanti dispari (utilizzate il numero 0 per indicare la fine della sequenza in ingresso).
- 1.6 Supponendo che l'esecutore conosca solo le operazioni di somma e sottrazione scrivete un algoritmo per il calcolo del prodotto di due numeri interi non negativi, x e y , ricevuti in ingresso, e un algoritmo per il calcolo del quoziente e del resto della divisione di x per y .
- 1.7 Supponendo che l'esecutore conosca le quattro operazioni scrivete un algoritmo per il calcolo della potenza x^y di due interi non negativi, x e y , ricevuti in ingresso. Riscrivete poi l'algoritmo nel caso in cui l'esecutore conosca solo le operazioni di somma e sottrazione.
- 1.8 Supponete che le variabili x e y contengano rispettivamente gli interi 4 e 10. Quali valori conterranno dopo avere eseguito l'assegnamento $x \leftarrow y$? E se invece venisse eseguito l'assegnamento $y \leftarrow x$?

1.11 Strutture di controllo fondamentali: esempi

Numeri pari e dispari

Vogliamo scrivere un algoritmo che, ricevendo un numero intero, indichi se esso è pari o dispari. La struttura dell'algoritmo si basa su una selezione:

```

leggi il numero n
calcola il resto della divisione di n per 2
SE il resto è uguale a zero
    ALLORA
        scrivi "pari"
    ALTRIMENTI
        scrivi "dispari"
FINESE

```

Riscriviamo ora l'algoritmo introducendo due variabili `n` e `resto`, destinate a contenere numeri interi. Indicheremo gli operatori binari per il calcolo del quoziente e del resto della divisione intera con `DIV` e `MOD`. Ad esempio $5 \text{ DIV } 3$ vale 1, mentre $5 \text{ MOD } 3$ vale 2.³ Utilizziamo inoltre il simbolo `==` per indicare l'operatore di confronto “uguale”.

variabili `n`, `resto`: numeri interi

```
leggi n
resto ← n MOD 2
SE resto == 0
  ALLORA
    scrivi "pari"
  ALTRIMENTI
    scrivi "dispari"
FINESE
```

Il calcolo del resto può essere effettuato direttamente al momento della valutazione della condizione della selezione, senza introdurre in modo esplicito una variabile:

variabili `n`: numero intero

```
leggi n
SE n MOD 2 == 0
  ALLORA
    scrivi "pari"
  ALTRIMENTI
    scrivi "dispari"
FINESE
```

Calcolo della somma di una sequenza di numeri

Descriveremo ora alcuni algoritmi per trattare iterativamente sequenze di numeri. Come primo esempio consideriamo il problema del calcolo della somma di una sequenza di numeri interi. Un primo schema di soluzione è questo:

```
poni il valore della somma a zero
ESEGUI
  leggi un numero
  aggiungi il numero alla somma
  QUANDO ci sono ancora numeri da leggere
    scrivi la somma
```

Come fa l'esecutore a sapere quando i numeri sono finiti? Un primo metodo consiste nel comunicare prima all'esecutore la quantità di numeri da leggere. L'esecutore ripete le istruzioni del ciclo tante volte quanti sono i numeri:

³ Per il calcolo della divisione si utilizza in Java l'operatore `/`, per il calcolo del resto l'operatore `%`.

```

leggi quanti numeri devi sommare
poni il valore della somma a zero
ESEGUI
    leggi un numero
    aggiungi il numero alla somma
QUANDO ci sono ancora numeri da leggere
    scrivi la somma

```

Osserviamo che l'algoritmo sopra richiede che ci sia sempre almeno un numero da sommare. Nel caso l'utente voglia sommare zero numeri, l'algoritmo non opera in maniera corretta.

Questo problema può essere risolto facilmente utilizzando un ciclo con il controllo in testa, cioè un ciclo del tipo QUANDO...RIPETI. Pertanto l'algoritmo può essere riscritto come:

```

leggi quanti numeri devi sommare
poni il valore della somma a zero
QUANDO ci sono ancora numeri da leggere ESEGUI
    leggi un numero
    aggiungi il numero alla somma
RIPETI
    scrivi la somma

```

Il controllo del ciclo precedente può essere realizzato introducendo una variabile per contare quanti numeri sono già stati letti. Tale variabile, posta a zero all'inizio dell'algoritmo, è incrementata a ogni iterazione. Introduciamo inoltre una variabile in cui immagazzinare la somma, una variabile per memorizzare l'ultimo numero letto, e una variabile, denominata quanti, dove memorizzare la quantità di numeri che devono essere letti:

```

variabili somma, numero, quanti, cont: numeri interi

leggi quanti
somma ← 0
cont ← 0
QUANDO cont < quanti ESEGUI
    leggi numero
    somma ← somma + numero
    cont ← cont + 1
RIPETI
    scrivi somma

```

In questa soluzione l'utente è costretto a comunicare a priori al programma quanti sono i numeri da sommare. Ciò può essere scomodo, soprattutto quando si ha un lungo elenco. Sarebbe più vantaggioso fissare un valore che indichi la fine della sequenza: utilizziamo a questo scopo il numero zero. Pertanto, se saranno inseriti 5 8 7 0, il risultato dovrà essere 20; se verrà invece inserito solamente 0, il risultato sarà 0. Basandoci sul primo schema di algoritmo che abbiamo presentato per questo problema possiamo scrivere:

poni il valore della somma a zero

ESEGUI

leggi un numero

aggiungi il numero alla somma

QUANDO il numero letto è diverso da zero

scrivi la somma

Osserviamo che il numero zero, che funge da indicatore di fine sequenza, è aggiunto alla somma prima di uscire dal ciclo. L'aggiunta di zero alla somma è inutile e, sebbene non modifichi il risultato, andrebbe evitata (ad esempio, se anziché calcolare la somma dovessimo calcolare il prodotto, la moltiplicazione per zero modificherebbe il risultato). Per fare ciò, possiamo introdurre un costrutto di selezione, in modo da eseguire l'operazione di somma solo quando il numero letto è diverso da zero:

poni il valore della somma a zero

ESEGUI

leggi un numero

SE il numero è diverso da zero**ALLORA**

aggiungi il numero alla somma

FINESE**QUANDO** il numero letto è diverso da zero

scrivi la somma

Notiamo che l'istruzione leggi un numero è eseguita sempre almeno una volta. Inoltre, a ogni ripetizione del ciclo, viene controllata due volte una condizione sul numero letto: la prima volta per la selezione, la seconda per l'iterazione. Per evitare questo doppio controllo possiamo riscrivere l'algoritmo come segue:

poni il valore della somma a zero

leggi un numero

QUANDO il numero è diverso da zero **ESEGUI**

aggiungi alla somma il numero

leggi un numero

RIPETI

scrivi la somma

Dettagliamo l'algoritmo introducendo due variabili, una per memorizzare la somma, l'altra per memorizzare l'ultimo numero letto. Indichiamo, come in Java, con != l'operatore di confronto diverso:

variabili somma, numero: numeri interi

somma ← 0

leggi numero

QUANDO numero != 0 **ESEGUI**

somma ← somma + numero

leggi numero

RIPETI
scrivi somma

Numeri pari e numeri dispari in una sequenza

Vogliamo ora descrivere un algoritmo che riceva dall'esterno una sequenza di numeri interi e indichi quanti di questi sono pari e quanti dispari. Supponiamo che l'inscrimento di zero indichi la fine della sequenza (zero non dovrà di conseguenza essere considerato come elemento della sequenza).

Osserviamo che l'ultimo algoritmo che abbiamo presentato è basato sulla seguente struttura:

...fase iniziale...
leggi un numero
QUANDO il numero è diverso da zero ESEGUI
 ...esamina il numero...
 leggi un numero
RIPETI
 ...fase finale...

Questa struttura può essere utilizzata ogni volta che si deve trattare una sequenza di numeri terminata da zero.

Nella ...fase iniziale... vengono assegnati valori alle quantità impiegate nell'algoritmo. In questo caso verranno utilizzati due contatori, il primo per i numeri pari, il secondo per i numeri dispari. Nella ...fase finale... i valori di questi contatori saranno comunicati all'esterno.

Analizziamo ora la fase ...esamina un numero... in cui si deve controllare se il numero è pari o dispari e incrementare il contatore opportuno:

SE il numero è pari
 ALLORA
 incrementa il contatore dei numeri pari
 ALTRIMENTI
 incrementa il contatore dei numeri dispari
FINESE

L'algoritmo completo è:

azzeri i due contatori
leggi un numero
QUANDO il numero è diverso da zero ESEGUI
 SE il numero è pari
 ALLORA
 incrementa il contatore dei numeri pari
 ALTRIMENTI
 incrementa il contatore dei numeri dispari
 FINESE
 leggi un numero
RIPETI
 scrivi i valori dei contatori

A questo punto l'algoritmo può essere dettagliato introducendo esplicitamente le variabili, le opportune istruzioni di assegnamento e le condizioni:

```

variabili numero, npari, ndispari: numeri interi
npari ← 0
ndispari ← 0
leggi numero
QUANDO numero != 0 ESEGUI
    SE numero MOD 2 == 0
        ALLORA
            npari ← npari + 1
        ALTRIMENTI
            ndispari ← ndispari + 1
    FINESE
    leggi numero
RIPETI
scrivi npari, ndispari

```

Calcolo del massimo comun divisore

Nel Paragrafo 1.1 abbiamo presentato l'algoritmo di Euclide per il calcolo del massimo comun divisore tra due interi positivi. Abbiamo poi visto come riscrivere questo algoritmo in un linguaggio assembler. Riscriviamo ora lo stesso algoritmo utilizzando le strutture di controllo fondamentali. Per comodità riportiamo il testo dell'algoritmo.

1. Siano x e y i due numeri.
2. Calcola il resto della divisione di x per y .
3. Se il resto è diverso da zero, ricomincia dal Passo 2
utilizzando come x il valore attuale di y , e come y il valore del resto,
altrimenti prosegui con il passo successivo.
4. Il massimo comun divisore è uguale al valore attuale di y .

Osserviamo che l'istruzione al Passo 2 dev'essere ripetuta un certo numero di volte sulla base di una condizione che è controllata al passo successivo. Dunque il controllo avviene *dopo* avere eseguito l'istruzione da ripetere. Inoltre, prima dell'esecuzione del ciclo, occorre effettuare la lettura dei dati, mentre al termine dell'esecuzione del ciclo è necessario comunicare all'esterno il risultato. Pertanto la struttura dell'algoritmo è:

```

leggi i numeri x, y
ESEGUI
...blocco di istruzioni...
QUANDO il resto è diverso da zero
scrivi il valore di y

```

Definiamo ora il ...blocco di istruzioni.... All'interno di esso si effettuerà il calcolo del resto della divisione. Qualora il resto sia diverso da zero, andranno predisposti i nuovi valori di x e di y da utilizzare nell'iterazione successiva secondo quanto specificato al Passo 2. Introduciamo un costrutto di selezione che effettua lo spostamento solo se il resto è diverso da zero.

L'alg

Risc
contNoti
il re
dell:
vari

calcola il resto della divisione di x per y
 SE il resto è diverso da zero

ALLORA

utilizza come x il valore attuale di y e
 come y il valore del resto

FINESE

L'algoritmo completo è quindi:

```
leggi i numeri  $x, y$ 
ESEGUI
  calcola il resto della divisione di  $x$  per  $y$ 
  SE il resto è diverso da zero
    ALLORA
      utilizza come  $x$  il valore attuale di  $y$  e
      come  $y$  il valore del resto
    FINESE
  QUANDO il resto è diverso da zero
    scrivi il valore di  $y$ 
```

Riscriviamo l'algoritmo introducendo tre variabili che chiameremo x , y , $resto$, destinate a contenere numeri interi.

variabili $x, y, resto$: numeri interi

```
leggi  $x, y$ 
ESEGUI
  resto  $\leftarrow x \text{ MOD } y$ 
  SE resto  $\neq 0$ 
    ALLORA
       $x \leftarrow y$ 
       $y \leftarrow resto$ 
    FINESE
  QUANDO resto  $\neq 0$ 
    scrivi  $y$ 
```

Notiamo che lo spostamento dei valori, dopo il calcolo del resto, può essere attuato anche quando il resto vale zero. In questo modo si evita di effettuare a ogni ciclo un doppio controllo sul valore della variabile $resto$. Tuttavia, a causa dello spostamento, il risultato finale si troverà nella variabile x :

variabili $x, y, resto$: numeri interi

```
leggi  $x, y$ 
ESEGUI
  resto  $\leftarrow x \text{ MOD } y$ 
   $x \leftarrow y$ 
   $y \leftarrow resto$ 
```

```
QUANDO resto != 0
scrivi x
```

1.12 Entità interagenti e oggetti: alcune idee

Il primo passo nella stesura di un algoritmo per la soluzione di un determinato problema consiste, di solito, nel domandarsi come si potrebbe risolvere lo stesso problema manualmente. La progettazione dell'algoritmo e del programma avviene dunque formalizzando e adattando all'esecutore meccanico i procedimenti che utilizziamo comunemente. Dunque gli algoritmi e i programmi possono essere visti come astrazioni o formalizzazioni, oppure anche come modelli di procedimenti impiegati nella realtà.

Finora abbiamo considerato algoritmi per risolvere problemi semplici e ben delimitati. In particolare abbiamo ipotizzato di avere un unico esecutore che, seguendo il procedimento descritto dall'algoritmo, sia in grado di risolvere il problema in esame.

Nella realtà un processo non è quasi mai il frutto di azioni svolte da un unico esecutore, ma è il risultato delle *interazioni* tra le attività svolte da agenti differenti.

Si consideri, per esempio, l'organizzazione di un ristorante. Possiamo individuare diverse *entità* o, meglio, *classi* di entità, che interagiscono: i camerieri, i cuochi, gli addetti alla cassa, i clienti. Ogni entità può interagire con altre entità per richiedere o per fornire un servizio. Ad esempio un cameriere, dopo avere raccolto l'ordinazione di un cliente, chiede a un cuoco la preparazione di determinati piatti. Possiamo osservare che:

- ogni entità è in grado di svolgere un determinato insieme di compiti, che dipendono dalla classe a cui l'entità appartiene (ad esempio tutti i camerieri sono in grado di svolgere gli stessi compiti, che sono diversi da quelli eseguiti dai cuochi); ogni compito verrà svolto eseguendo un determinato algoritmo;
- il comportamento complessivo del sistema è il risultato del coordinamento e delle interazioni tra le varie entità che lo costituiscono.

La programmazione a oggetti permette di descrivere un sistema complesso a partire dalle entità che lo costituiscono, gli *oggetti*, definendo il loro modo di interagire.

Poiché in un sistema possono esserci più entità in grado di fornire gli stessi comportamenti, detti *metodi*, gli oggetti sono raggruppati in *classi* (ad esempio tutti i camerieri appartengono alla stessa classe: sono capaci di raccogliere le ordinazioni e di servire i clienti; ciò che differenzia un cameriere da un altro è l'insieme dei clienti da servire). Più precisamente, gli oggetti vengono costruiti a partire dalle classi:

- si definisce la classe descrivendo i dati e i *metodi*, cioè i comportamenti degli oggetti; in questo senso possiamo immaginare che la classe definisca *come sono fatti gli oggetti* o che la classe sia una “fabbrica” di oggetti;
- successivamente è possibile definire oggetti appartenenti alla classe, detti anche *istanze della classe*;

- oggetti della stessa classe, o di classi differenti, possono interagire tra loro mediante lo scambio di messaggi o, in altre parole, mediante l'invocazione di metodi.

Tutti questi aspetti, che adesso possono apparire astratti e oscuri, saranno sviluppati e concretizzati in seguito, durante lo studio del linguaggio Java.

Esercizi

- 1.9** Scrivete un algoritmo, basato sulle tre strutture di controllo fondamentali, che riceva in ingresso una sequenza di numeri e produca in uscita il prodotto dei numeri letti. Supponete che l'inserimento del numero zero indichi la fine della sequenza.
- 1.10** Scrivete un algoritmo, basato sulle tre strutture di controllo fondamentali, che riceva in ingresso una sequenza di numeri interi e produca in uscita la somma dei numeri pari e la somma dei numeri dispari contenuti nella sequenza. Supponete che l'inserimento del numero zero indichi la fine della sequenza.
- 1.11** Scrivete un algoritmo, basato sulle tre strutture di controllo fondamentali, che riceva in ingresso una sequenza di numeri interi e produca in uscita la somma dei numeri di posto pari e la somma dei numeri di posto dispari contenuti nella sequenza. Supponete che l'inserimento del numero zero indichi la fine della sequenza.
- 1.12** Scrivete un algoritmo, basato sulle tre strutture di controllo fondamentali, che riceva in ingresso una sequenza di numeri e produca in uscita la media dei numeri letti. Supponete che l'inserimento del numero zero indichi la fine della sequenza.
- 1.13** Scrivete un algoritmo, basato sulle tre strutture di controllo fondamentali, che riceva in ingresso una sequenza di numeri interi e riporti in uscita il più grande e il più piccolo tra i numeri letti. Supponete che l'inserimento del numero zero indichi la fine della sequenza.
- 1.14** Scrivete un algoritmo, basato sulle tre strutture di controllo fondamentali, per il calcolo del prodotto di due fattori interi positivi. Impiegate la seguente tecnica: a ogni passo si raddoppia il primo fattore e si dimezza il secondo, terminando quando il secondo fattore diventa 1. Il prodotto dei due fattori è dato dalla somma dei primi fattori corrispondenti a secondi fattori dispari. Ad esempio, per il calcolo del prodotto tra 34 e 21, si effettueranno i seguenti passi:

primo fattore	secondo fattore	numero da sommare
34	21	34
68	10	
136	5	136
272	2	
544	1	544

La somma dei valori della terza colonna è uguale al prodotto dei due numeri dati.

1.15 Analizzate più in dettaglio l'esempio del ristorante descrivendo per ognuna delle classi di oggetti individuate le possibili interazioni con le altre classi.

1.13 Grammatiche

Per definire la sintassi di un linguaggio occorre specificarne la grammatica, cioè l'insieme delle regole per la costruzione delle frasi o sentenze del linguaggio.

Una grammatica G è definita da questi quattro elementi:

- (1) un insieme finito T di *simboli terminali*, cioè di simboli che costituiranno le sentenze del linguaggio;
- (2) un insieme finito N di *simboli non terminali*, o *metasimboli*, utilizzati nella costruzione delle sentenze del linguaggio;
- (3) un insieme finito P di *regole di produzione*; per i nostri scopi ci limitremo a considerare grammatiche in cui le regole di produzione specificano come un metasimbolo possa essere sostituito da una sequenza di simboli terminali e metasimboli;
- (4) un *simbolo iniziale* S , appartenente all'insieme dei simboli non terminali, utilizzato come punto di partenza nella costruzione delle sentenze.

Il linguaggio generato dalla grammatica G è l'insieme di tutte le sequenze di simboli terminali ottenibili applicando le regole di produzione dell'insieme P , a partire dal simbolo iniziale S .

Esempio

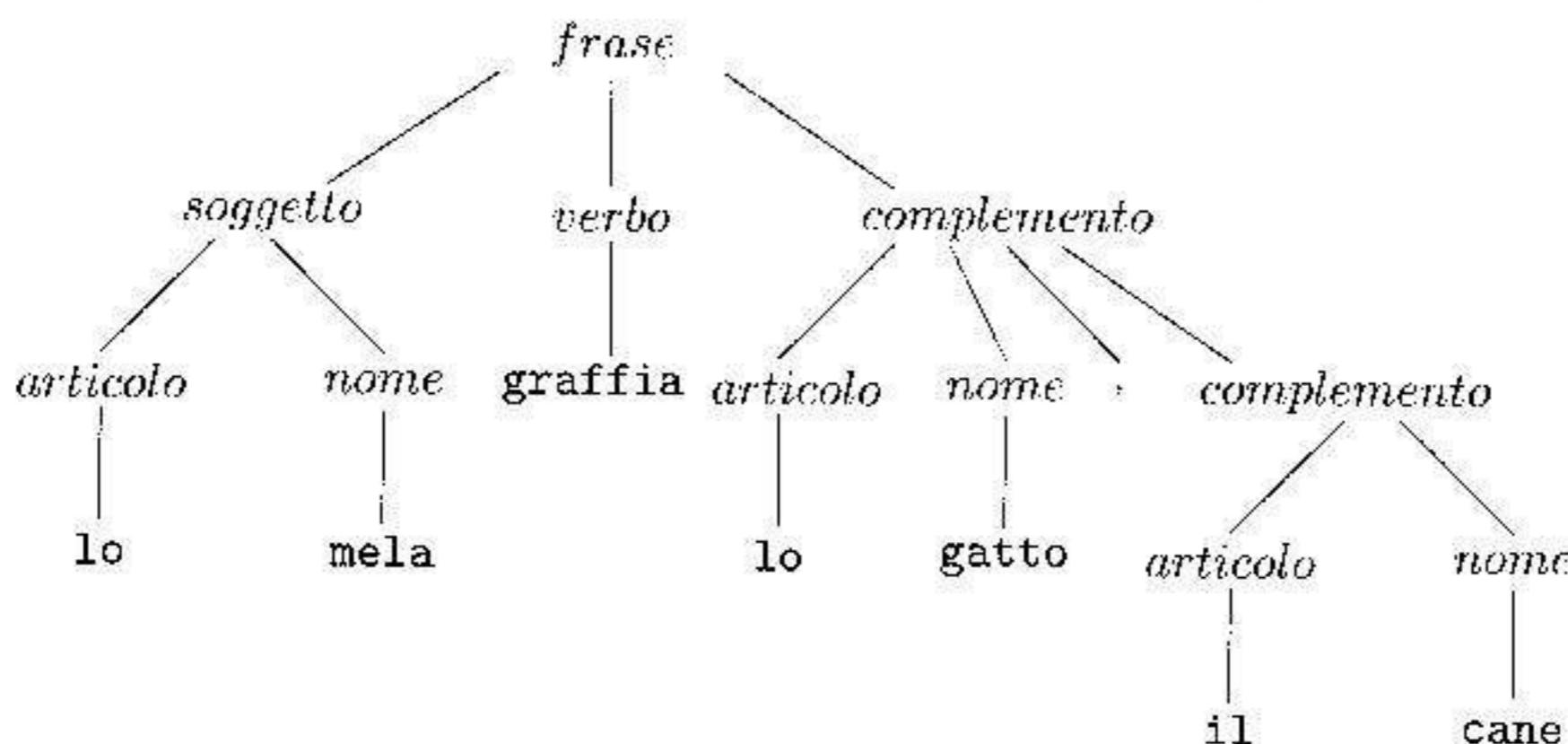
Si consideri la grammatica $G = (T, N, P, S)$ definita come segue.

- $T = \{\text{il}, \text{lo}, \text{la}, \text{cane}, \text{mela}, \text{gatto}, \text{mangia}, \text{graffia}, ,\}$
- $N = \{\text{frase}, \text{soggetto}, \text{verbo}, \text{complemento}, \text{articolo}, \text{nome}\}$
- P è l'insieme formato dalle seguenti regole espresse in BNF (forma di Backus-Naur):
 - $\text{frase} ::= \text{soggetto} \text{ verbo} \text{ complemento}$
 - $\text{soggetto} ::= \text{articolo} \text{ nome}$
 - $\text{articolo} ::= \text{il} \mid \text{la} \mid \text{lo}$
 - $\text{nome} ::= \text{cane} \mid \text{mela} \mid \text{gatto}$
 - $\text{verbo} ::= \text{mangia} \mid \text{graffia}$
 - $\text{complemento} ::= \text{articolo} \text{ nome} \mid \text{articolo} \text{ nome} , \text{ complemento}$

La parte destra di ciascuna regola specifica come dev'essere trasformato il metasimbolo indicato sulla sinistra. Il simbolo | indica un'alternativa. Ad esempio l'ultima regola va letta come: *complemento* è un *articolo* seguito da un *nome*, oppure un *articolo* seguito da un *nome* seguito dal simbolo , seguito da un *complemento*.

- $S = frase.$

Lo schema seguente mostra la costruzione della sentenza *lo mela graffia lo gatto, il cane* a partire da *frase*, applicando le regole di produzione indicate sopra.

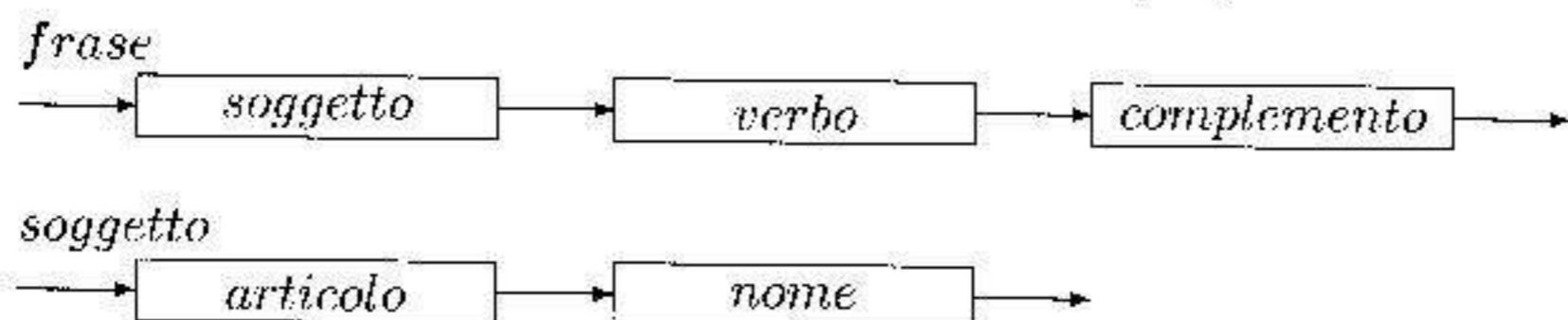


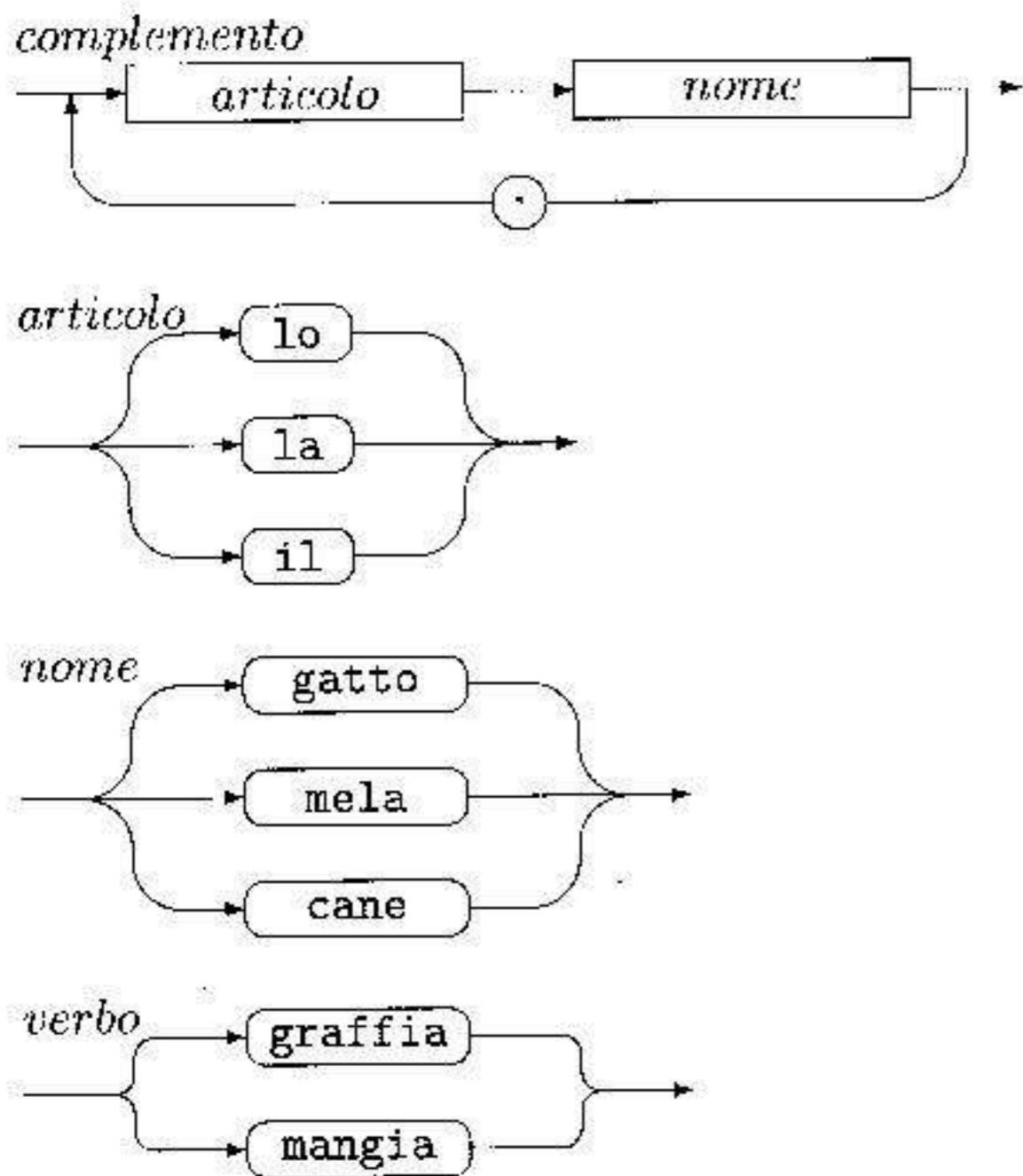
È possibile seguire anche il procedimento inverso, ossia partire da una sequenza di simboli terminali e verificare se sia una sentenza del linguaggio risalendo al simbolo iniziale mediante le regole di produzione.

Un compilatore conosce la grammatica del linguaggio e applica questo procedimento per verificare la correttezza sintattica dei programmi.

Carte sintattiche

In alternativa alla forma BNF, le regole di produzione possono essere espresse per mezzo di particolari diagrammi denominati *carte sintattiche*. In particolare, si specifica una carta sintattica per ciascun simbolo non terminale della grammatica. In una carta sintattica i rettangoli indicano simboli non terminali (che andranno espansi con le carte sintattiche corrispondenti), mentre gli ovali indicano simboli terminali, che quindi non devono essere espansi ulteriormente. Ogni biforcazione indica un'alternativa. A titolo d'esempio riportiamo le carte sintattiche corrispondenti alle produzioni della grammatica introdotta nell'esempio precedente.





1.14 La grammatica del linguaggio Java: il lessico

Introduciamo ora gli elementi lessicali della grammatica del linguaggio Java, cioè, in sostanza, l’insieme dei simboli terminali utilizzati per costruire programmi Java.

- *Alfabeto.*

L’alfabeto utilizzato per scrivere i programmi si chiama *Unicode*, ed è un insieme di caratteri rappresentati su 16 bit che contiene tutte le lettere dell’alfabeto inglese, tutte le cifre e i simboli usuali che si trovano sulla tastiera, come +, -, *, /, (. Sono presenti poi molti alfabeti utilizzati nelle diverse lingue del mondo⁴.

- *Parole riservate (o parole chiave).*

Sono parole che nel linguaggio hanno un significato predeterminato. Non possono essere utilizzate diversamente e non possono essere ridefinite.

⁴ Lo standard Unicode, inizialmente a 16 bit, è stato esteso per permettere la rappresentazione di ulteriori caratteri. I dettagli di questa estensione non vengono trattati in quanto esulano dagli scopi di questo testo.

Le parole riservate di Java sono:

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

- *Identificatori.*

Sono nomi impiegati all'interno del programma per indicare variabili, classi, riferimenti a oggetti, e così via. Un identificatore è costituito da una sequenza di lettere e cifre che inizia con una lettera. Oltre alle lettere dell'alfabeto latino sono consentite anche lettere di altri alfabeti; inoltre, si possono utilizzare i caratteri *underscore* (_) e dollaro (\$). È importante sottolineare che, a differenza di quanto avviene per altri linguaggi, il compilatore Java distingue le lettere maiuscole dalle minuscole. Pertanto gli identificatori pippo, Pippo e PIPPO sono diversi.

- *Separatori.*

Sono caratteri che permettono di separare o raggruppare parti di codice. I separatori utilizzati nel linguaggio Java sono:

() { } [] ; , .

- *Operatori.*

Sono simboli o sequenze di simboli che denotano alcune operazioni. Gli operatori di Java sono:

=	>	<	!	~	?	:	
==	<=	>=	!=	&&		++	--
+	-	*	/	&		^	%
+=	-=	*=	/=	&=	=	^=	%=
						<<	>>
						<<=	>>=
							>>>=

- *Letterali.*

Sono sequenze di caratteri utilizzate all'interno dei programmi per rappresentare valori di tipi primitivi e stringhe. Ad esempio la sequenza di caratteri 1234 rappresenta un numero intero, mentre le sequenze "questa è una stringa" e "1234" rappresentano stringhe di caratteri. I letterali saranno illustrati in dettaglio più avanti, insieme con i tipi corrispondenti.

- *Commenti.*

Esistono due tipi di commenti:

- *commenti delimitati dai caratteri /* e */*: il compilatore ignora tutto il testo compreso tra questi caratteri; tali commenti possono estendersi per più righe;
- *commenti a fine riga*: si aprono con la coppia di caratteri // e si chiudono alla fine della riga; il compilatore ignora il testo che inizia dai caratteri // fino alla fine della riga.

I commenti del primo tipo, che si aprono con la sequenza di caratteri /**, sono detti *commenti di documentazione*. Esiste un programma, chiamato javadoc, che genera automaticamente la documentazione relativa a un programma Java estraendo le informazioni contenute nei commenti di documentazione. Descriveremo in modo circostanziato questo tipo di documentazione in seguito.

Non forniremo una definizione formale dell'intera sintassi del linguaggio Java, ma evidenzieremo la sintassi o la struttura dei singoli costrutti man mano che saranno introdotti.

1.15 Il primo programma Java

Di solito ogni libro di testo che introduce un linguaggio di programmazione presenta come primo esempio un breve programma che visualizza un messaggio. Lo faremo anche noi per iniziare a chiarire alcuni elementi della sintassi di Java e per fornire un esempio su cui sperimentare il funzionamento del compilatore e della Java Virtual Machine. Prima di passare all'esemplificazione, premettiamo che non sarà possibile spiegare subito, in dettaglio, il significato di tutti gli elementi che compaiono nel programma. Alcuni di essi potranno essere compresi solo successivamente, quando si disporrà di maggiori conoscenze.

Ecco il testo del primo programma:

```
class BuonInizio {
    public static void main(String [] args) {
        System.out.println("Ti auguro una buona giornata!");
    }
}
```

Un programma Java è costituito da un insieme di *classi*. Nei primi esempi che faremo, utilizzeremo un'unica classe. Le classi si aprono con la parola riservata `class`, seguita da un identificatore scelto dal programmatore, il nome della classe, in questo caso `BuonInizio`. I nomi delle classi iniziano con una lettera maiuscola. Il testo di questo programma dovrà essere scritto in un file chiamato `BuonInizio.java`, cioè dovrà essere scritto con il nome della classe seguito dall'estensione `.java`. Il compilatore produrrà il bytecode corrispondente a questo programma e lo memorizzerà in un file di nome `BuonInizio.class`, pronto per essere eseguito dalla Java Virtual Machine.

La prima riga del programma rappresenta l'*intestazione* della classe, che in questo caso specifica solo il nome della classe. La riga termina con una parentesi graffa aperta, che è chiusa nell'ultima riga del programma. Le due parentesi graffe contengono la *definizione* della classe. Questa classe contiene specificatamente un metodo di nome `main`. Il significato delle parole `public static void`, che precedono il nome del metodo, sarà spiegato più avanti. Dopo il nome del metodo vengono indicati, tra parentesi tonde, i *parametri* del metodo. Anche il significato dei parametri utilizzati qui sarà chiarito in seguito. Per ora ci limiteremo a segnalare che l'esecuzione di un programma avviene sempre a partire da un metodo di nome `main`, con un'intestazione simile a quella indicata in quest'esempio. La riga d'intestazione termina con una parentesi graffa aperta, che, nell'esempio viene chiusa nella penultima riga del programma. All'interno di questa coppia di parentesi si trova il codice del metodo. Ogni istruzione è terminata dal carattere punto e virgola. L'unica istruzione di questo programma è:

```
System.out.println("Ti auguro una buona giornata!");
```

Quest'istruzione è l'invocazione di un metodo `println` dell'oggetto `System.out`.

Nella programmazione a oggetti la computazione avviene tramite interazioni tra oggetti. Ogni oggetto è capace, mediante i metodi, di svolgere determinati compiti. È possibile richiedere a un oggetto lo svolgimento di un servizio (cioè l'esecuzione di un metodo) utilizzando un *riferimento all'oggetto* seguito dal nome del servizio, vale a dire dal nome del metodo richiesto, seguito da ulteriori dettagli relativi al servizio. In questo esempio `System.out` è un riferimento (definito all'interno della libreria di Java) allo schermo; `println` è il nome di un servizio che lo schermo è in grado di eseguire; la stringa tra parentesi fornisce all'oggetto che deve eseguire il servizio ulteriori informazioni relative al servizio richiesto. In altre parole, l'istruzione richiede al video di eseguire il metodo `println`, il cui effetto è quello di visualizzare sullo schermo la stringa specificata tra parentesi, muovendo poi il cursore a capo riga. È importante pensare il video come un oggetto autonomo, in grado di svolgere alcuni servizi (metodi) su richiesta. Il riferimento all'oggetto, in questo caso a `System.out`, permette di rivolgere la richiesta. Dopo il riferimento all'oggetto, e dopo un punto per la separazione, si indica il servizio richiesto tramite un *messaggio* all'oggetto. Il messaggio è costituito dal nome del metodo (nell'esempio `println`) seguito da informazioni aggiuntive (gli *argomenti*, cioè le informazioni indicate tra parentesi), utili allo svolgimento del servizio (nell'esempio la stringa di caratteri "Ti auguro una buona giornata!").

L'oggetto `System.out` è in grado di eseguire anche altri servizi: il metodo `print`, con una stringa come argomento, visualizza la stringa specificata *senza* riportare a capo il cursore; il metodo `println`, senza argomenti, riporta a capo il cursore e non scrive nulla. Per esempio l'effetto prodotto dalla riga di codice indicata sopra è uguale all'effetto che si ottiene con le seguenti quattro invocazioni di metodi di `System.out`:

```
System.out.print("Ti ");
System.out.print("auguro una");
System.out.print(" buona giornata!");
System.out.println();
```

Parte I

Uso degli oggetti

Protocolli e contratti

Tutti i linguaggi di programmazione forniscono meccanismi di astrazione. Un linguaggio assembler, che costituisce il livello di astrazione più elementare, offre semplicemente una rappresentazione simbolica delle istruzioni del processore. I linguaggi di programmazione imperativi, come Pascal, C e FORTRAN, forniscono invece astrazioni più o meno raffinate della macchina sottostante; in particolare i linguaggi imperativi “ad alto livello” mettono a disposizione strutture di controllo che sono astrazioni delle istruzioni di controllo disponibili in assembler, tipi primitivi che sono astrazioni dei tipi disponibili in assembler e meccanismi per raffinare tali astrazioni (definizione di nuovi tipi e nuove istruzioni).

Sebbene queste astrazioni costituiscano un notevole passo avanti rispetto a quelle fornite dal linguaggio assembler, restano ancora saldamente legate alla struttura del calcolatore. Dovendo sviluppare un programma per risolvere un problema, il programmatore è obbligato a progettare e a realizzare il programma in termini della struttura del calcolatore sottostante anziché nei termini della struttura del problema. In sostanza, il linguaggio fornisce un *mondo* o *dominio della soluzione* (il mondo del calcolatore) in cui il programmatore deve simulare gli elementi che compaiono nel dominio del problema introducendo opportune traduzioni; la soluzione del problema è quindi realizzata manipolando tali rappresentazioni tramite le istruzioni del linguaggio. Non a caso, uno dei passi fondamentali nella progettazione di un programma imperativo consiste nell'individuare la rappresentazione dei *dati* rilevanti per la soluzione del problema. La difficoltà di realizzare la corrispondenza fra dominio del problema e dominio della soluzione dà luogo a soluzioni innaturali difficilmente comprensibili in termini di dominio del problema; questo ha un grosso impatto sul processo di produzione e di manutenzione dei programmi.

I linguaggi di programmazione a oggetti, come ad esempio Java, forniscono invece astrazioni che consentono di rappresentare direttamente nel dominio della soluzione gli elementi che compaiono nel dominio del problema. In questo modo la corrispondenza fra i due domini può essere espressa chiaramente: a ogni *agente*, cioè a ogni *oggetto* che compare nel dominio del problema, corrisponde un analogo *oggetto software* nel dominio della soluzione; il programmatore può quindi cercare di risolvere il problema simulando nel dominio della soluzione il modo in cui risolverebbe il problema nel mondo reale.

Abbiamo parlato di agenti (gli oggetti del mondo reale) e di oggetti (la loro rappresentazione in un linguaggio di programmazione) in termini intuitivi, senza darne una definizione. Ovvia-

mente, dato che un linguaggio di programmazione dovrà mettere a disposizione una struttura sintattica per descrivere tali rappresentazioni (gli oggetti), bisogna stabilire una volta per tutte quali siano le caratteristiche rilevanti per la rappresentazione di un oggetto del mondo reale. L'assunzione alla base del paradigma programmazione a oggetti è che le caratteristiche salienti per la rappresentazione di un oggetto siano il suo *stato* e il suo *comportamento*.

Lo *stato* di un oggetto è l'insieme delle proprietà che caratterizzano l'oggetto in un determinato istante. Ad esempio un'automobile è un oggetto caratterizzato da proprietà come il colore, la marca, la quantità di carburante, la velocità, etc. Evidentemente alcune di queste informazioni non cambiano nel tempo; ad esempio la marca dell'automobile non cambierà per tutta l'esistenza dell'oggetto automobile. Altre invece cambiano. Osservando infatti l'oggetto automobile in momenti diversi potremmo riscontrare valori differenti della sua velocità, della marcia innestata, del livello del carburante.

Il *comportamento* è invece l'insieme delle azioni che contraddistingue l'oggetto. Per esempio, un'automobile può frenare, accelerare, cambiare marcia, rifornirsi, etc. Naturalmente per far eseguire l'azione all'oggetto automobile dovremo interagire con esso. Così per accelerare dovremo premere il pedale destro, mentre per frenare dovremo premere quello centrale. L'oggetto reale automobile offre pertanto una serie di dispositivi meccanici tramite i quali possiamo comunicare con l'automobile. Astraendo dai dispositivi meccanici coinvolti, possiamo immaginare che le varie azioni (frenare, accelerare, e così via) abbiano inizio inviando un opportuno *messaggio* all'automobile. L'insieme dei messaggi che un oggetto è in grado di interpretare prende il nome di *interfaccia* o *protocollo*. L'interfaccia di un oggetto non cambia nel tempo, mentre può cambiare il modo in cui un oggetto reagisce a un certo messaggio. Ad esempio il modo in cui un'automobile reagisce alla pressione del pedale dell'acceleratore è diverso a seconda che il motore sia acceso oppure spento.

I linguaggi di programmazione a oggetti mettono a disposizione costrutti che consentono di definire *oggetti* (qui intesi come oggetti software per distinguere dagli oggetti del mondo reale) specificandone lo stato e il comportamento. Un programma si basa sulla collaborazione di un certo numero di oggetti.

Per capire che cosa intendiamo per collaborazione, osserviamo in che modo un problema può essere risolto nel mondo reale tramite una collaborazione fra oggetti. Supponiamo che il nostro problema sia quello di ordinare una torta per una festa di compleanno. Un modo per risolverlo è quello di recarci alla nostra pasticceria di fiducia, che supponiamo si chiami *daPinoPasticcino*, e fare l'ordinazione al commesso. Indipendentemente dalle parole che utilizziamo per fare l'ordinazione, noi stiamo *invia*ndo un *messaggio* a un *agente* (il commesso), in cui i dati rilevanti sono il tipo di torta e il giorno in cui passeremo a ritirarla. Astraendo dai dettagli irrilevanti, possiamo schematizzare il nostro messaggio in questo modo:

```
ordinaTorta(torta,data)
```

in cui *ordinaTorta* identifica il tipo di messaggio, mentre *torta* e *data* sono due argomenti che specificano le informazioni rilevanti del messaggio. Possiamo schematizzare questa interazione con il commesso della pasticceria nel modo descritto nella Figura 2.1, dove sono evidenziati i due agenti coinvolti nell'interazione e il messaggio inviato dall'agente *cliente* all'agente



Figura 2.1 Il messaggio e gli oggetti coinvolti.

daPinoPasticcino (qui per semplicità stiamo identificando il commesso della pasticceria con la pasticceria stessa). In generale, l'effetto della nostra azione sarà quello di ottenere la torta per la data indicata.

Inviando il messaggio con la nostra ordinazione al commesso facciamo due supposizioni fondamentali: la prima è che l'agente che abbiamo selezionato sia in grado di riconoscere il messaggio, la seconda è che l'agente sia in grado di soddisfare la nostra richiesta, cioè che l'effetto della nostra richiesta sia quello di produrre la torta specificata per la data fissata. Nel caso dei linguaggi a oggetti, l'insieme dei messaggi che un oggetto è in grado di riconoscere prende il nome di *interfaccia*, mentre il significato di un messaggio, cioè l'azione che l'agente compie in risposta al messaggio, costituisce il *contratto* di tale messaggio. Un altro concetto importante è quello di *responsabilità*. Quando un oggetto mette a disposizione un messaggio nella sua interfaccia, esso si assume la responsabilità di soddisfare il contratto. Si osservi che le nozioni di interfaccia, contratto e responsabilità sono alla base di tutte le cooperazioni che ci coinvolgono nel mondo reale. Nel nostro esempio abbiamo individuato un *agente* appropriato (daPinoPasticcino), ossia un agente che nella sua interfaccia prevede un messaggio il cui contratto ci consente di risolvere il nostro problema (avere la torta). Gli abbiamo *invia*to un messaggio contenente la nostra richiesta, e ci aspettiamo che il contratto venga rispettato, cioè che l'agente si assuma la responsabilità di mantenere ciò che promette.

Nel mondo reale la nostra conoscenza dell'interfaccia di un agente e dei contratti deriva dalla nostra conoscenza del mondo. Inoltre, sia nel caso dell'interfaccia sia in quello dei contratti è ammesso un certo grado di imprecisione e variabilità. Nel caso di un linguaggio di programmazione, invece, perché un oggetto possa essere utilizzato nel modo corretto, interfacce e contratti dovranno essere chiaramente documentati.

Tornando all'esempio osserviamo che, una volta che abbiamo ordinato la torta al commesso, non siamo generalmente interessati ai dettagli relativi al modo in cui l'azione richiesta verrà realizzata. Dal nostro punto di vista, l'unica cosa importante è l'effetto del nostro messaggio. Dal punto di vista del commesso, invece, soddisfare la nostra richiesta comporterà l'avvio di una sequenza di operazioni. La sequenza di passi che viene iniziata da un agente in risposta a un messaggio prende il nome di *metodo*. Tale sequenza di azioni può coinvolgere solo l'agente cui abbiamo inviato il messaggio, oppure può coinvolgerne altri. Ad esempio il commesso della pasticceria potrebbe mettere l'ordinazione in uno schedario. Tale schedario potrebbe essere giornalmente controllato dal pasticcere, che al momento giusto preleverà l'ordinazione dalla coda e inizierà la preparazione della torta collaborando a sua volta con altri agenti, come ad esempio i suoi aiutanti, gli strumenti di cucina, la dispensa, e via dicendo.

Dall'esempio descritto possiamo dedurre un primo principio della programmazione a oggetti:

- nella programmazione a oggetti un'azione è iniziata inviando un *messaggio* a un *oggetto*

responsabile di svolgere l’azione

- il *messaggio* codifica la richiesta di un’azione ed è corredata dell’informazione necessaria (*argomenti*) a soddisfare la richiesta
- il *ricevente*, se accetta il messaggio, si assume la *responsabilità* di portare a termine la relativa azione rispettando il contratto. Ciò sarà fatto eseguendo il *metodo* corrispondente.

Il concetto di *responsabilità* è fondamentale in questo approccio: nel momento in cui inviamo un messaggio a un agente assumiamo che il contratto promesso sia rispettato. Dal punto di vista della programmazione questo significa che presumiamo che l’implementazione fornita dall’oggetto cui chiediamo un servizio sia corretta.

Sempre sulla base del nostro esempio, osserviamo che il metodo che abbiamo individuato per risolvere il problema non è l’unico possibile. Invece di entrare in una pasticceria avremmo potuto chiedere alla nostra segretaria di fare l’ordinazione per noi. Anche la nostra segretaria avrebbe potuto avere nella sua interfaccia il medesimo messaggio, cioè avrebbe potuto essere in grado di accettare il messaggio `ordinaTorta(torta, data)` e di garantire il contratto che prevede di farci avere la torta per la data fissata. Ovviamente il metodo che la segretaria avrebbe messo in atto una volta ricevuto il messaggio sarebbe stato diverso da quello seguito dal commesso della pasticceria `daPinoPasticcino`. È molto probabile che la segretaria avrebbe risolto il problema telefonando alla pasticceria `daPinoPasticcino` e inviando il medesimo messaggio al commesso della pasticceria. Analogamente, nella programmazione a oggetti esistono oggetti diversi che possono rispondere allo stesso messaggio.

Osserviamo ora che, se invece di rivolgerci alla pasticceria `daPinoPasticcino` fossimo entrati in una pasticceria a caso, non sarebbe cambiato nulla per quel che riguarda il messaggio e il contratto. Infatti sappiamo che ogni *Pasticceria* è in grado di accettare il messaggio `ordinaTorta(torta, data)` per il solo fatto di appartenere alla categoria *Pasticceria*. Nell’ambito della programmazione a oggetti, il corrispondente di una *categoria* di agenti che condividono il medesimo *stato* e *comportamento* prende il nome di *classe*. Nell’esempio la nostra pasticceria di fiducia (`daPinoPasticcino`) è una delle possibili istanze della categoria *Pasticceria*. Le istanze della categoria condividono un’interfaccia comune. Una classe è quindi un prototipo che specifica le informazioni presenti nello stato e il comportamento di tutte le sue istanze (oggetti).

In questo capitolo presenteremo il modo in cui Java implementa i concetti che abbiamo introdotto intuitivamente a partire da un semplice esempio di collaborazione fra agenti nel mondo reale. In particolare, in questo capitolo impareremo a realizzare semplici applicazioni costruendo oggetti da classi definite, e facendoli collaborare fra loro unicamente sulla base della descrizione delle interfacce e dei contratti.

2.1 Primi esempi di oggetti e classi in Java

Come abbiamo detto in precedenza, le caratteristiche di un oggetto (stato e comportamento) sono descritte da una classe (la categoria cui appartiene l’oggetto). In Java le classi sono l’elemento

base dei programmi: il programmatore può solo scrivere classi; non è possibile scrivere codice al di fuori di una classe. Le classi in Java costituiscono quindi l'aspetto statico della programmazione; quando scriveremo il codice di un programma scriveremo classi. In particolare sarà nelle classi che definiremo i metodi, cioè l'insieme dei messaggi cui un oggetto della classe può rispondere e il modo in cui lo fa. In fase di esecuzione la soluzione di un problema è definita facendo interagire oggetti ottenuti costruendo istanze delle classi. Nei prossimi paragrafi svilupperemo alcune semplici applicazioni facendo collaborare fra loro oggetti costruiti a partire da classi predefinite di Java e da alcune classi sviluppate appositamente per questo testo.

La classe `ConsoleOutputManager`

La classe `ConsoleOutputManager` è una classe sviluppata per questo testo. Gli oggetti di tale classe realizzano canali di comunicazione con il dispositivo di output standard, cioè con il video.¹ La classe mette a disposizione metodi che consentono di visualizzare a video vari tipi di dati; ad esempio, a un oggetto di tipo `ConsoleOutputManager` possiamo inviare il messaggio `println("Ecco il mio primo programma")`, che ha come effetto la visualizzazione della stringa "Ecco il mio primo programma" sul video. Per poter inviare un messaggio a un oggetto della classe `ConsoleOutputManager` dovremo prima creare tale oggetto.

Per costruire gli oggetti, Java mette a disposizione la parola riservata `new`, un operatore che dev'essere seguito dall'invocazione del *costruttore* della classe di cui si vuole creare l'oggetto. Un costruttore di una classe è una parte di codice, definita all'interno della classe, che si occupa di inizializzare in modo opportuno lo stato dell'oggetto. Per il momento non ci interessa conoscere nei dettagli com'è fatto un costruttore e come agisce, ma ci interessa esclusivamente utilizzarlo. A tale scopo è sufficiente sapere che i costruttori hanno sempre lo stesso nome della classe e potrebbero prevedere degli argomenti. Per costruire un oggetto dovremo utilizzare un'*espressione* con la seguente sintassi:

`new nome_della_classe (lista_argomenti)`

Ad esempio la classe `ConsoleOutputManager` mette a disposizione un costruttore che non richiede argomenti. Possiamo pertanto costruire un oggetto di questa classe utilizzando l'espressione:

`new ConsoleOutputManager()`

Osserviamo che abbiamo chiamato questa frase del linguaggio Java *espressione*, e non *istruzione*. Le espressioni sono in sostanza sequenze di operatori e di operandi costruite secondo le regole sintattiche del linguaggio. In questo caso l'operatore è la parola chiave `new`: si tratta di un operatore unario (cioè con un solo argomento) e prefisso (cioè che precede il proprio argomento). Le espressioni hanno un tipo complessivo e danno luogo, in fase di esecu-

¹ È possibile comunicare con il dispositivo di output standard anche tramite il riferimento predefinito `System.out`, come abbiamo fatto nell'esempio del Paragrafo 1.15. Tuttavia l'uso della classe `ConsoleOutputManager`, richiedendoci di introdurre l'istruzione esplicita per la creazione dell'oggetto corrispondente al canale di comunicazione, è utile per ricordare che il dispositivo di output è utilizzato dai programmi come un qualunque altro oggetto.

zione, a un *valore*. In particolare le *espressioni di creazione* che stiamo considerando hanno come tipo la classe dell’oggetto costruito e, in fase di esecuzione, producono come valore un *riferimento a un oggetto della classe specificata dal costruttore*. In fase di esecuzione, l’espressione `new ConsoleOutputManager()` produce un riferimento a un oggetto di tipo `ConsoleOutputManager`. Ovviamente il riferimento prodotto da questa espressione dovrà essere memorizzato da qualche parte per poter essere utilizzato in seguito. A questo scopo utilizziamo una variabile, che chiamiamo `video`, dichiarata di tipo `ConsoleOutputManager`. In particolare possiamo scrivere l’istruzione:

```
ConsoleOutputManager video = new ConsoleOutputManager();
```

che, oltre all’invocazione del costruttore, contiene una *dichiarazione di variabile* e un *assegnamento*. Analizziamola nei minimi particolari.

- A sinistra del simbolo `=` è dichiarata una variabile di tipo `ConsoleOutputManager` e di nome `video`: questo significa che da questo punto in avanti la variabile `video` può essere utilizzata per indicare un oggetto della classe `ConsoleOutputManager` o, più precisamente, può contenere un *riferimento* a un oggetto della classe `ConsoleOutputManager`.
- A destra del simbolo `=` è richiamato il costruttore della classe `ConsoleOutputManager`; la parola riservata `new`, che precede il nome del costruttore, indica la richiesta di fabbricare un’istanza della classe.
- Infine, mediante l’operatore `=`, l’oggetto così costruito è assegnato alla variabile `video`. Più precisamente, il risultato dell’espressione `new ConsoleOutputManager()` è un riferimento all’oggetto costruito. L’assegnamento memorizza questo riferimento nella variabile `video`.

A questo punto, ogni volta che chiederemo all’oggetto costruito di svolgere un servizio, cioè ogni volta che vorremo richiamarne un metodo, ci rivolgeremo a esso utilizzando la variabile `video`. La situazione nella memoria della macchina dopo l’esecuzione dell’istruzione precedente è riassunta nella Figura 2.2. La variabile `video` e l’istanza di `ConsoleOutputManager` corrispondono a zone diverse della memoria. `video` è una variabile che identifica una zona della memoria che contiene come valore un *riferimento* a un oggetto. In sostanza, `video` contiene l’informazione sull’indirizzo cui possono essere recapitati i messaggi che vogliamo inviare all’oggetto. Per ora non abbiamo invece idea del tipo di informazioni memorizzate in un oggetto. Come vedremo in seguito, la memoria riservata a un oggetto contiene le informazioni relative al suo stato e alcune informazioni che servono alla Java Virtual Machine per la gestione dell’oggetto. Per ora ci interessa soltanto sapere che fra queste informazioni di controllo è compreso il nome della classe a partire dalla quale l’oggetto è stato costruito; per questo motivo nella Figura 2.2 abbiamo evidenziato tale informazione.

Vediamo ora come si invia un messaggio a un oggetto. Quest’operazione prende il nome di *invocazione di metodo* e ha la seguente sintassi:

riferimento_a_oggetto . nome_methodo (lista_argomenti)

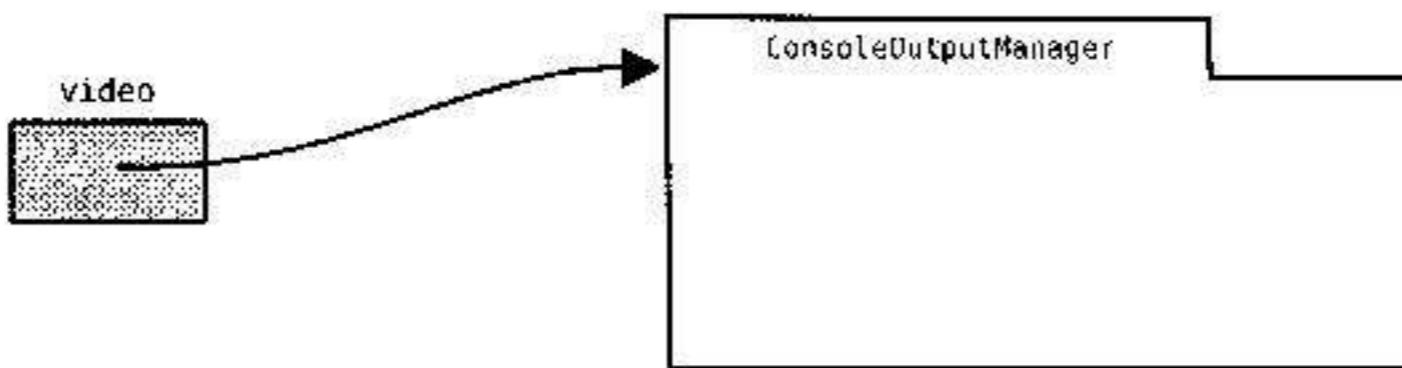


Figura 2.2 Variabile riferimento e oggetto.

Supponiamo di voler chiedere all'oggetto che rappresenta il video del nostro computer di visualizzare una stringa; possiamo utilizzare la seguente istruzione:

```
video.println("Ecco il mio primo programma");
```

in cui il riferimento all'oggetto è la variabile `video` e il nome del metodo è `println`. Questo metodo riceve come argomento la stringa da visualizzare. Riassumendo, le istruzioni utilizzate per stampare a video la stringa "Ecco il mio primo programma" sono:

```
ConsoleOutputManager video = new ConsoleOutputManager();
video.println("Ecco il mio primo programma");
```

Si osservi che le istruzioni in Java sono sempre concluse dal carattere `;`. Queste due linee di codice non costituiscono un'applicazione eseguibile. Infatti, come abbiamo accennato in precedenza, in Java non è possibile scrivere codice al di fuori delle classi. In particolare, per poter eseguire un programma in Java è necessario scrivere almeno una classe che contenga un metodo di nome `main` utilizzato dalla Java Virtual Machine come punto di partenza per l'esecuzione di qualunque programma. Spieghiamo ora come scrivere un'applicazione eseguibile; non sarà tuttavia possibile spiegare subito, in dettaglio, il significato di tutti gli elementi che la compongono. Ecco il testo dell'applicazione:

```
import prog.io.ConsoleOutputManager;

class PrimoProgramma {
    public static void main(String[] args) {
        ConsoleOutputManager video = new ConsoleOutputManager();
        video.println("Ecco il mio primo programma!");
    }
}
```

In questo programma riconosciamo le istruzioni che abbiamo descritto in precedenza; il resto del codice (a parte la prima linea) descrive la struttura della classe che manderemo in esecuzione. La prima linea:

```
import prog.io.ConsoleOutputManager;
```

è una *direttiva di importazione*, e indica al compilatore che nel codice che segue utilizzeremo la classe `ConsoleOutputManager` del package `prog.io`. Come spiegheremo meglio più avanti, la direttiva di importazione fornisce al compilatore le informazioni necessarie a reperire il file `ConsoleOutputManager.class` contenente il bytecode della classe. Questo file contiene tutte le informazioni sulla classe `ConsoleOutputManager` necessarie al compilatore per compilare il programma e alla Java Virtual Machine per eseguirlo. Il resto del codice descrive la classe `PrimoProgramma`. La parola riservata `class` indica che stiamo definendo una nuova classe il cui nome (`PrimoProgramma`) è l'identificatore scelto dal programmatore. Per convenzione i nomi delle classi sono scritti con l'iniziale maiuscola. Attenersi a questa convenzione migliora la leggibilità dei programmi. Vediamo ora come procedere per compilare ed eseguire questo programma.

Compilazione ed esecuzione

Come abbiamo detto, la direttiva `import` indica al compilatore e alla Java Virtual Machine dove trovare il file `.class` che contiene tutte le informazioni necessarie per utilizzare la classe. L'argomento della direttiva di importazione, nel nostro caso `prog.io.ConsoleOutputManager`, indica al compilatore che il file `ConsoleOutputManager.class` fa parte di un *package*, cioè di una libreria, denominata `prog.io`. Il compilatore e la Java Virtual Machine interpretano i punti che compaiono in `prog.io.ConsoleOutputManager` come dei separatori di directory che indicano dove trovare il file `ConsoleOutputManager.class`. In ambienti Unix e Linux il separatore fra le directory è costituito ad esempio dal carattere `/`²; tale direttiva informa il compilatore che i dettagli relativi alla classe `ConsoleOutputManager` si trovano in un file di nome

```
prog/io/ConsoleOutputManager.class
```

Quando il compilatore ha bisogno di informazioni su questa classe, va a cercare il file identificato da `prog/io/ConsoleOutputManager.class` a partire da una serie di directory che sono specificate in una variabile di sistema denominata `CLASSPATH`, che di solito contiene la lista delle directory in cui archiviamo le librerie di classi. Per maggiori informazioni sulla variabile `CLASSPATH` si veda l'Appendice C.³

La prima cosa che dovremo fare per compilare ed eseguire il nostro programma sarà scrivere il testo in un file di nome `PrimoProgramma.java`.⁴ Il nome del file deve avere obbligatoriamente l'estensione `.java`, altrimenti il compilatore segnalerà che gli è impossibile trovare il file da compilare (per una presentazione degli errori più comuni si veda l'Appendice C). A questo punto possiamo compilare il programma impartendo il comando

```
javac PrimoProgramma.java
```

² Nel caso dei sistemi DOS e Windows il separatore fra le directory è `\`.

³ Le classi che appartengono alle librerie possono anche essere fornite mediante file in formato JAR (Java ARchive). Affinché risultino accessibili, la posizione di questi file deve essere specificata mediante la variabile `CLASSPATH`.

⁴ Come vedremo in seguito, non è obbligatorio in questo caso utilizzare come nome del file il nome della classe. È buona norma comunque usare questa convenzione per organizzare meglio i file contenenti le classi.

Per effetto di questo comando, se il programma è stato scritto correttamente, il compilatore genera un file di nome `PrimoProgramma.class` nella medesima directory in cui è stata effettuata la compilazione. Il file contenente il bytecode può essere eseguito dalla Java Virtual Machine, richiamabile tramite il comando `java`. Quindi per eseguire l'applicazione dobbiamo impartire il comando:

```
java PrimoProgramma
```

L'effetto dell'esecuzione è il seguente:

```
> java PrimoProgramma
Ecco il mio primo programma!
```

Si osservi che il comando per l'esecuzione comprende solo il nome della classe, e non l'estensione `.class` che il compilatore attribuisce al file contenente il bytecode.

La classe `ConsoleInputManager`

Anche la classe `ConsoleInputManager`, di cui presenteremo ora un esempio d'uso, è stata sviluppata appositamente per questo testo e non fa parte della distribuzione standard di Java. La ragione per la quale abbiamo sviluppato queste classi per l'input/output non è dovuta alla mancanza di analoghe classi in Java, che anzi fornisce un package per la gestione dell'input/output estremamente potente e flessibile (tale package è descritto nel Capitolo 13). Il fatto è che l'uso di questo package richiederebbe di introdurre fin da ora molti concetti fondamentali del linguaggio che non potrebbero essere adeguatamente descritti in questa parte del testo e che non sono richiesti per le classi di input/output sviluppate per il testo; esse risultano però meno flessibili e complete, anche se sufficienti per la maggior parte degli utilizzi comuni.

Come la classe `ConsoleOutputManager` consente di realizzare canali di comunicazione verso il video, così la classe `ConsoleInputManager` consente di realizzare canali di lettura dalla tastiera. Al pari dell'esempio d'uso di `ConsoleOutputManager` dato in precedenza, per leggere dati da tastiera dovremo per prima cosa costruire un oggetto che rappresenta la tastiera, salvandone il riferimento in una variabile di tipo opportuno, che utilizzeremo poi come destinatario dei messaggi di lettura. In particolare, fra i messaggi che possiamo inviare a un oggetto di tipo `ConsoleInputManager` ci sono il messaggio `readLine`, che permette di leggere una riga di testo, e `readInt`, che permette di leggere un numero intero.

Per illustrare l'utilizzo delle classi appena introdotte, presentiamo come primo esempio una semplice applicazione il cui compito è quello di leggere una riga inserita dall'utente mediante la tastiera e di riscrivere la stessa riga a video. Una riga di testo non è altro che una sequenza di caratteri, cioè una *stringa*. Per trattare le stringhe Java mette a disposizione la classe `String` definita nel package `java.lang`. Questo package contiene un insieme di classi fondamentali per la programmazione in Java; per questa ragione è importato automaticamente. Non è quindi necessario utilizzare la direttiva di importazione per la classe `String`.

Anche in questo caso dovremo per prima cosa costruire gli oggetti che ci servono. In particolare avremo bisogno di un canale di comunicazione con il video, fornito dalle istanze della

classe `ConsoleOutputManager`, e di un canale di comunicazione con la tastiera, fornito dalle istanze della classe `ConsoleInputManager`. Dato che la classe `ConsoleInputManager` offre un costruttore privo di argomenti, possiamo costruire i due oggetti con le istruzioni:

```
ConsoleOutputManager video = new ConsoleOutputManager();
ConsoleInputManager tastiera = new ConsoleInputManager();
```

In questo caso i riferimenti agli oggetti sono stati memorizzati, rispettivamente, nelle variabili `video`, dichiarata di tipo `ConsoleOutputManager`, e `tastiera`, dichiarata di tipo `ConsoleInputManager`. Per leggere la stringa dalla tastiera possiamo utilizzare l'espressione:

```
tastiera.readLine()
```

che invoca il metodo `readLine` dell'oggetto cui fa riferimento `tastiera`. Osserviamo che a differenza dei metodi `println` utilizzati nell'esempio precedente, il metodo `readLine` produce un risultato: la stringa letta o, meglio, un riferimento a un oggetto della classe `String` contenente tale stringa. Per poter utilizzare la stringa letta in un secondo tempo, occorre memorizzarne il riferimento in una variabile di tipo `String` in questo modo:

```
String messaggio = tastiera.readLine();
```

A questo punto dobbiamo riportare sul video la stringa letta. A tale scopo chiediamo all'oggetto riferito dalla variabile `video` di eseguire il proprio metodo `println`, fornendogli come argomento la variabile `messaggio` che contiene il riferimento alla stringa da visualizzare.

```
video.println(messaggio)
```

Il metodo `println` non restituisce alcun valore; ciò viene indicato dicendo che il tipo del suo risultato è `void`. Pertanto l'invocazione precedente costituisce da sola un'istruzione che va terminata, come sempre, con il punto e virgola.

Il programma complessivo si ottiene includendo nel corpo del metodo `main` di una classe che chiameremo `Pappagallo` le istruzioni descritte sopra. All'inizio del programma dovremo inoltre importare le classi `ConsoleOutputManager` e `ConsoleInputManager`. Ecco il testo completo della classe così costruita:

```
import prog.io.ConsoleOutputManager;
import prog.io.ConsoleInputManager;

class Pappagallo {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager tastiera = new ConsoleInputManager();
        ConsoleOutputManager video = new ConsoleOutputManager();

        //lettura e comunicazione
        String messaggio = tastiera.readLine();
```

```

        video.println(messaggio);
    }
}

```

Osserviamo che l'applicazione precedente, per quanto semplice, svolge il suo compito instaurando una collaborazione fra tre oggetti, identificati nel programma dalle variabili `tastiera`, `video` e `messaggio`.

Notiamo anche che, nell'applicazione precedente, possiamo sostituire le due direttive di importazione:

```

import prog.io.ConsoleOutputManager;
import prog.io.ConsoleInputManager;

```

con la sola direttiva:

```
import prog.io.*;
```

La direttiva di importazione che utilizza `*` prende il nome di *importazione su richiesta* e sarà discussa a fondo nel Capitolo 7. Per ora possiamo pensare che il significato di questa direttiva sia quello di importare tutte le classi che compaiono nel package `prog.io`. Ovviamente, mentre l'importazione su richiesta risulta più comoda da utilizzare, l'altra forma di importazione mette in evidenza le reali dipendenze del codice che stiamo scrivendo. Per comodità utilizzeremo l'importazione su richiesta per le classi del package `prog.io`, mentre importeremo in modo esplicito eventuali altre classi.

Scriviamo ora una variante della classe precedente: dopo aver chiesto all'utente di inserire il proprio nome, sul video è riportato un messaggio di saluto contenente il nome inserito.

```

import prog.io.*;

class Saluto {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager tastiera = new ConsoleInputManager();
        ConsoleOutputManager video = new ConsoleOutputManager();

        //lettura del nome da tastiera
        String nome = tastiera.readLine("Ciao, come ti chiami? ");

        //comunicazione
        video.print("Buongiorno ");
        video.print(nome);
        video.println("!");
    }
}

```

Questa volta, diversamente dall'esempio precedente, nell'invocazione del metodo `readLine` è stato fornito un argomento (si tratta di un letterale stringa). Mentre nel caso precedente il metodo si limitava a leggere una stringa di caratteri inserita dall'utente, in questo caso il metodo riporta sul video la stringa fornita come argomento prima di effettuare la lettura. Questo è estremamente utile nella scrittura di programmi interattivi per guidare l'utente nell'inserimento dei dati. Anche se in ambedue i casi lo scopo di `readLine` è lo stesso, le operazioni eseguite sono diverse: in effetti siamo in presenza di due metodi differenti che hanno lo stesso nome. In base alla presenza o all'assenza di argomenti, possiamo riconoscere quale dei due metodi `readLine` è invocato.

Anche per la classe `ConsoleOutputManager` abbiamo utilizzato un nuovo metodo: il metodo `print` che, come il metodo `println`, riceve come argomento una stringa. Mentre `println` riporta la stringa sul video e poi sposta il cursore a capo, il metodo `print` lascia il cursore dopo l'ultimo carattere scritto. Pertanto una successiva operazione di scrittura riprenderà dalla stessa riga.

Segnaliamo, infine, che la classe `ConsoleOutputManager` contiene un altro metodo di nome `println` che non richiede argomenti. Il compito di questo metodo è quello di spostare il cursore dalla riga in cui si trova all'inizio della riga successiva. Per una presentazione dettagliata dei metodi e delle classi del package `prog.io` si veda la documentazione del package `prog` contenuta nel CD-ROM allegato al testo.

2.2 Prototipi e segnature

Negli esempi precedenti abbiamo visto come creare un oggetto, memorizzarne il riferimento e utilizzare quest'ultimo per invocare un metodo. In particolare abbiamo visto che per invocare un metodo nel modo corretto è necessario conoscerne il *prototipo*, cioè il nome, la lista degli *argomenti* con il relativo tipo e il tipo del *risultato* restituito. Per utilizzare un metodo in modo adeguato è anche necessario, naturalmente, conoscerne il *contratto*, ossia ciò che il metodo fa.

Consideriamo ad esempio il metodo `print` della classe `ConsoleOutputManager`. Questo metodo riceve come argomento una stringa e non restituisce alcun risultato; ciò si indica anche dicendo che il tipo restituito dal metodo è `void`. La tabella seguente mette in evidenza i prototipi di alcuni metodi della classe `ConsoleOutputManager`:

<i>tipo restituito</i>	<i>nome del metodo</i>	<i>argomenti</i>
<code>void</code>	<code>print</code>	<code>int</code>
<code>void</code>	<code>print</code>	riferimento a <code>String</code>
<code>void</code>	<code>println</code>	nessuno
<code>void</code>	<code>println</code>	<code>int</code>
<code>void</code>	<code>println</code>	riferimento a <code>String</code>

Si noti che la classe `ConsoleOutputManager` contiene diversi metodi con il medesimo nome, ma con lista degli argomenti diversa.

D'ora in poi sarà importante avere ben presente la distinzione fra prototipo e segnatura di un metodo:

- la *segnatura* di un metodo è costituita dal nome del metodo e dai tipi dei suoi argomenti;
- il *prototipo* di un metodo è costituito dal tipo del valore restituito dal metodo, dal nome del metodo e dai tipi dei suoi argomenti (quindi *prototipo=tipo restituito + segnatura*). Se il metodo non restituisce nulla, in luogo del tipo restituito si utilizza la parola riservata `void`.

Nella tabella sono presenti tre metodi che hanno il medesimo nome `println`, ma segnature diverse: in particolare il primo metodo `println` è privo di argomenti, il secondo riceve un argomento di tipo `int` (come vedremo si tratta di un tipo di Java per la rappresentazione di numeri interi), mentre il terzo riceve come argomento un riferimento a un oggetto di tipo `String`.

La possibilità di avere metodi con lo stesso nome ma segnatura diversa prende il nome di *overloading* (letteralmente "sovraffollamento"⁵). L'uso dell'overloading è molto comune nei linguaggi a oggetti, ed evita di introdurre nomi artificiali per metodi che hanno funzioni simili pur agendo su argomenti diversi.

Dal punto di vista del contratto, il primo metodo `println` manda semplicemente il cursore all'inizio della linea successiva del video, mentre gli altri due stampano a video il loro argomento prima di mandare il cursore all'inizio della linea successiva. Si tratta quindi di metodi diversi, anche se con effetti simili (in particolare la sequenza di operazioni eseguita in risposta al messaggio è diversa), invocati mediante messaggi con il medesimo nome ma liste di argomenti (segnature) diverse.

Quando viene invocato un metodo il cui nome è sovraccaricato, il compilatore è in grado di capire quale metodo intendiamo invocare in base alla lista degli argomenti che forniamo al metodo. Se, ad esempio, abbiamo dichiarato `video` di tipo `ConsoleOutputManager` e scriviamo l'istruzione:

```
video.println("Ciao");
```

il compilatore sa che intendiamo invocare il secondo metodo della tabella, perché l'argomento è di tipo `String`. Non è invece possibile avere in una classe metodi con la stessa segnatura ma tipo restituito diverso. Ad esempio tutti i metodi di nome `println` della tabella precedente restituiscono `void`; pertanto `ConsoleOutputManager` non può contenere metodi con questo nome e con una delle liste di argomenti indicate in precedenza, che restituiscono un tipo diverso.

Anche la classe `ConsoleInputManager`, di cui riassumiamo qui sotto i prototipi di alcuni metodi, fa ampio uso dell'overloading.

<i>tipo restituito</i>	<i>nome del metodo</i>	<i>argomenti</i>
riferimento a <code>String</code>	<code>readLine</code>	nessuno
riferimento a <code>String</code>	<code>readLine</code>	riferimento a <code>String</code>
<code>int</code>	<code>readInt</code>	nessuno
<code>int</code>	<code>readInt</code>	riferimento a <code>String</code>

⁵ Il termine deriva dal fatto che al medesimo nome sono associati più significati (i diversi metodi).

In particolare tale classe mette a disposizione metodi per leggere vari tipi di valori dalla tastiera. Nella tabella sono evidenziati i metodi per leggere un `int` (un intero) e una stringa nelle due versioni disponibili: quella priva di argomenti, che legge semplicemente un `input` da tastiera, e quella con un argomento di tipo `String`, che visualizza la `String` specificata come messaggio per guidare l'inserimento dei dati da parte dell'utente.

Presentiamo ora un programma che visualizza la somma di due interi inseriti da tastiera. Ricordiamo che la somma può essere calcolata mediante questo semplice algoritmo, che utilizza tre variabili, `i`, `j` e `k`, in grado di contenere numeri interi:

```
leggi i
leggi j
k ← i + j
scrivi k
```

Il metodo che presentiamo richiede lo svolgimento delle operazioni di lettura e scrittura, come descritto in precedenza. Le variabili `i`, `j` e `k` sono dichiarate all'inizio del codice di `main`, subito dopo l'intestazione. La somma di due variabili di tipo `int` si ottiene tramite l'operatore di somma `+`, scrivendo `i + j`, che è un'espressione di tipo `int` (cioè un'espressione il cui valore è di tipo `int`).

```
/* Calcola la somma di due interi letti da tastiera. */
import prog.io.*;

class Somma {
    public static void main(String[] args) {
        int i, j, k; //variabili utilizzate per dati e risultati

        //predisposizione dei canali di comunicazione
        ConsoleInputManager tastiera = new ConsoleInputManager();
        ConsoleOutputManager video = new ConsoleOutputManager();

        //lettura dei dati
        i = tastiera.readInt();
        j = tastiera.readInt();

        //calcolo della somma
        k = i + j;

        //comunicazione del risultato
        video.println(k);
    }
}
```

L'esecuzione inizia, come sempre, dal metodo `main`. Dopo la dichiarazione delle variabili e la creazione degli oggetti per la lettura e la scrittura, si hanno i due assegnamenti alle variabili `i` e

`j` dei valori letti da tastiera. Per ottenere tali valori, il programma richiede all'oggetto `tastiera` di eseguire il metodo `readInt`. A questo punto l'esecuzione resta sospesa finché l'utente non inserisce un dato. Le variabili `i` e `j` sono state dichiarate di tipo `int`, che è esattamente il tipo restituito dal metodo `readInt`. Ovviamente i valori inseriti dall'utente dovranno essere sequenze di cifre. Se così non fosse, il metodo non potrebbe rispettare il proprio contratto che garantisce che il valore restituito sia di tipo `int`. Per garantire il proprio contratto, il metodo `readInt` della classe `ConsoleInputManager` chiede all'utente di reinserire il valore fino a quando l'utente non inserisce un numero intero.

Nell'applicazione precedente, per agevolare la comprensione del codice, sono stati introdotti alcuni commenti. Osserviamo che il primo commento fornisce una breve descrizione dell'applicazione, mentre i commenti all'interno del metodo `main` forniscono indicazioni sul significato delle variabili e dei gruppi di istruzioni. Lo scopo dei commenti è quello di dare un'indicazione sul significato delle istruzioni che agevoli la comprensione del codice e del modo in cui è organizzato. Il modo di commentare il codice ha un grande impatto sulla modificabilità e manutenzione del codice stesso. Per comodità nel prosieguo eviteremo di introdurre il commento iniziale che descrive il compito dell'applicazione, sempre indicato nel testo, mentre utilizzeremo costantemente i commenti per fornire una linea guida per la comprensione del codice. È importante che anche il lettore impari subito a commentare le applicazioni che scriverà.

Il programma precedente è estremamente laconico: non informa l'utente su quali sono i valori attesi in `input` e su quale sia il loro significato in relazione al programma. È sempre opportuno invece che un programma guidi l'utente nell'inserimento dei dati, visualizzando messaggi di richiesta. Analogamente, è utile che il risultato venga fornito insieme con un breve messaggio. Riscriviamo, a questo scopo, il programma utilizzando i metodi `readInt` nella versione con una stringa come argomento; inoltre espandiamo la fase di comunicazione del risultato in modo che vengano segnalati sia i valori dei dati inseriti sia quello del risultato ottenuto:

```
import prog.io.*;

class Somma {
    public static void main(String[] args) {
        int i, j, k; //variabili utilizzate per dati e risultati

        //predisposizione dei canali di comunicazione
        ConsoleInputManager tastiera = new ConsoleInputManager();
        ConsoleOutputManager video = new ConsoleOutputManager();

        //lettura dei dati
        i = tastiera.readInt("Inserisci il primo numero da sommare: ");
        j = tastiera.readInt("Inserisci il secondo numero da sommare: ");

        //calcolo della somma
        k = i + j;
```

```

    //comunicazione del risultato
    video.print(i);
    video.print(" + ");
    video.print(j);
    video.print(" fa ");
    video.println(k);
}
}

```

Se, ad esempio, vengono inseriti i valori 4 e 37, il messaggio visualizzato sarà:

4 + 37 fa 41

Utilizzando i metodi e gli operatori relativi agli oggetti di tipo `String`, che introduciamo nel prossimo paragrafo, le ultime istruzioni del metodo `main` possono essere scritte in forma più compatta.

2.3 La classe `String`

L'altra classe che abbiamo utilizzato negli esempi precedenti è `String`. Come abbiamo detto, si tratta di una classe i cui oggetti modellano stringhe, cioè sequenze arbitrarie di caratteri. In Java le stringhe non sono soltanto dati, ma oggetti, e sono quindi caratterizzate da stato e comportamento. Come al solito il comportamento è descritto dai metodi della classe, mentre lo stato dell'oggetto memorizza la stringa rappresentata dall'oggetto stesso.

La classe `String` mette a disposizione un costruttore che riceve come argomento una sequenza di caratteri compresa fra doppi apici. Quindi, per creare un oggetto `String` che rappresenta la stringa "Java" e memorizzare il suo riferimento in una variabile `s` possiamo utilizzare l'istruzione:

```
String s = new String("Java");
```

In Java le sequenze di caratteri comprese fra doppi apici prendono il nome di *letterali di tipo String*, e consentono di specificare valori di tipo `String` all'interno di un programma Java.⁶ Dato che nei programmi l'uso di oggetti di tipo `String` è molto comune, Java fornisce un meccanismo隐式的 per creare oggetti di questo tipo ricorrendo semplicemente al letterale, senza dover invocare esplicitamente il costruttore; ad esempio, l'istruzione:

```
String s = "Java";
```

ha lo stesso effetto della precedente.

Osserviamo ancora che, poiché i doppi apici hanno un significato speciale in quanto rappresentano l'inizio e la fine di un letterale di tipo `String`, non possiamo utilizzarli direttamente in

⁶ In generale i *letterali* rappresentano i valori di un tipo all'interno di un programma Java. La classe `String` è l'unica classe per la quale Java prevede dei letterali. Il linguaggio prevede anche letterali per i tipi primitivi e il letterale `null` per le variabili riferimento.

una stringa. È però possibile utilizzarli facendoli precedere dal carattere \, che li priva del loro significato speciale. Quindi l'istruzione:

```
String s = "\\";
```

ha l'effetto di costruire un oggetto di tipo `String`, il quale modella la stringa che consiste del solo carattere \.

Presentiamo ora alcuni metodi della classe `String`, indicandone prima i prototipi e discutendone quindi il contratto.

<i>tipo restituito</i>	<i>nome del metodo</i>	<i>argomenti</i>
riferimento a <code>String</code>	<code>toUpperCase</code>	nessuno
riferimento a <code>String</code>	<code>toLowerCase</code>	nessuno
int	<code>length</code>	nessuno
riferimento a <code>String</code>	<code>concat</code>	riferimento a <code>String</code>
riferimento a <code>String</code>	<code>substring</code>	due int
riferimento a <code>String</code>	<code>substring</code>	un int

Il primo metodo della classe `String` che consideriamo è il metodo `toUpperCase`. Tale metodo restituisce come risultato il riferimento a una nuova stringa costituita dagli stessi caratteri della stringa che esegue il metodo, con l'eccezione delle lettere minuscole, che sono trasformate nelle maiuscole corrispondenti.

Ad esempio, tramite le istruzioni:

```
String s1 = "Ciao";
String s2 = s1.toUpperCase();
```

è assegnato a `s1` il riferimento all'oggetto che modella la stringa "Ciao", quindi a `s2` viene assegnato il riferimento all'oggetto ottenuto invocando il metodo `toUpperCase` dell'oggetto cui si riferisce `s1`. Dunque `s2` conterrà un riferimento all'oggetto che rappresenta la stringa "CIAO". Non è necessario memorizzare il riferimento a un oggetto per poter invocare un suo metodo: dato che l'effetto dell'invocazione del costruttore (implicito) di `String` restituisce un riferimento a un oggetto, è possibile utilizzare subito questo riferimento come destinatario di un messaggio. Ad esempio, l'istruzione:

```
String s2 = "Ciao".toUpperCase();
```

ha anch'essa l'effetto di assegnare a `s2` un oggetto che modella la stringa "CIAO": diversamente dall'esempio precedente, non potremo inviare altri messaggi all'oggetto "Ciao" per il semplice fatto che non ne abbiamo conservato il riferimento in una variabile.

Possiamo riscrivere la precedente classe `Pappagallo` in maniera che il messaggio inserito dall'utente sia visualizzato trasformando le minuscole in maiuscole:

```
import prog.io.ConsoleOutputManager;
import prog.io.ConsoleInputManager;
```

```

class Pappagallo {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager tastiera = new ConsoleInputManager();
        ConsoleOutputManager video = new ConsoleOutputManager();

        //lettura del messaggio e costruzione della risposta
        String messaggio = tastiera.readLine();
        String risposta = messaggio.toUpperCase();

        //comunicazione
        video.println(risposta);
    }
}

```

Si noti la richiesta fatta all'oggetto `messaggio` di eseguire il proprio metodo `toUpperCase` al fine di ottenere un riferimento alla stringa da visualizzare. Tale riferimento è assegnato alla variabile `risposta`.

Un aspetto importante degli oggetti di tipo `String` è che essi *non sono modificabili*. Per tutta la sua esistenza, un oggetto di tipo `String` modella sempre la stessa stringa. Ad esempio, se nel programma precedente la stringa inserita dall'utente è "Ciao", l'invocazione di metodo `messaggio.toUpperCase()` non modifica la stringa ma restituisce un riferimento a un *nuovo* oggetto di tipo `String`, che modella la stringa "CIAO". La situazione in memoria dopo l'invocazione del metodo `messaggio.toUpperCase()` è descritta nella Figura 2.3.

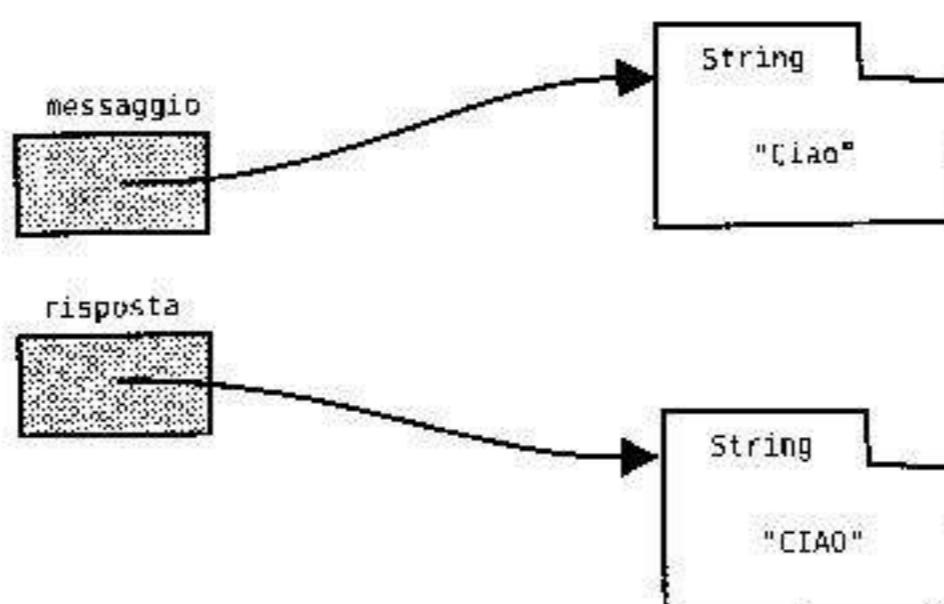


Figura 2.3 Situazione in memoria dopo l'esecuzione di `messaggio.toUpperCase()`.

Il metodo `toLowerCase` è analogo al metodo illustrato sopra, e restituisce il riferimento a una stringa in cui le lettere maiuscole della stringa che esegue il metodo sono trasformate in minuscole.

Il metodo `length` restituisce invece un `int`, cioè un numero intero, uguale alla lunghezza della stringa rappresentata dall'oggetto che lo esegue. Ad esempio, l'invocazione:

`"Ciao".length()`

restituisce come risultato 4. Anche il letterale `""` è un valore ammissibile per un oggetto di tipo `String`: esso rappresenta la *stringa vuota*. In particolare,

`"".length()`

è un'invocazione del metodo `length` dell'oggetto che rappresenta la stringa vuota; tale invocazione restituisce come risultato 0.

Il metodo `concat` restituisce un riferimento alla stringa ottenuta concatenando alla stringa **che esegue il metodo** la stringa fornita come argomento. Ad esempio, se `s1` e `s2` sono state dichiarate come riferimenti a `String`, l'invocazione

`s1.concat(s2)`

restituisce il riferimento alla stringa ottenuta concatenando la stringa riferita da `s2` (che compare come argomento) alla stringa riferita da `s1` (cui è richiesta l'esecuzione del metodo). Le ultime tre istruzioni del metodo `main` della classe `Saluto` potrebbero essere sostituite dalle seguenti:

```
String risposta = "Buongiorno ".concat(nome).concat("!");
video.println(risposta);
```

oppure:

```
String risposta = "Buongiorno ".concat(nome.concat("!"));
video.println(risposta);
```

Nel primo caso si chiede alla stringa "Buongiorno " di costruire una nuova stringa concatenando a sé la stringa riferita da `nome`; si chiede poi alla stringa risultante di concatenare a sé la stringa "!". Il riferimento alla stringa così ottenuta è assegnato alla variabile `risposta`, utilizzata nell'istruzione successiva come argomento nell'invocazione del metodo `println`. Si poteva anche evitare di introdurre la variabile `risposta` scrivendo, più brevemente:

```
video.println("Buongiorno ".concat(nome).concat("!"));
```

Nel secondo caso, sebbene il risultato sia lo stesso, il procedimento è diverso. Infatti si richiede alla stringa "Buongiorno" di concatenare a sé il risultato della concatenazione della stringa "`nome`" con la stringa "!".

Anche la concatenazione di stringhe è un'operazione molto comune nei programmi. Per questa ragione Java consente di esprimere in modo più comodo mediante l'operatore `+`. Ad esempio possiamo costruire la stringa `risposta` così:

```
String risposta = "Buongiorno " + nome + "!";
```

Non è possibile scrivere letterali di tipo `String` che occupino più di una riga; ad esempio, l'istruzione

```
String messaggio = "Questo è un messaggio particolarmente lungo che
dobbiamo spezzare in più righe";
```

non viene accettata dal compilatore. Possiamo però spezzare questa stringa in più righe utilizzando in modo opportuno l'operatore `+`, come segue:

```
String messaggio = "Questo è un messaggio particolarmente lungo che " +
    "dobbiamo spezzare in più righe";
```

L'operatore `+` può essere applicato anche a una stringa e a un numero intero. In questo caso, per calcolare il risultato della concatenazione, il valore intero è trasformato nella stringa di cifre che lo rappresenta. La parte di comunicazione del risultato del metodo `main` della precedente classe `Somma` può quindi essere riscritta come:

```
video.println(i + " + " + j + " fa " + k);
```

I due metodi `substring` permettono di estrarre porzioni, o *sottostringhe*, della stringa. Prima di descrivere questi metodi occorre precisare che in una stringa le posizioni dei caratteri vengono contate a partire da zero. Si consideri, ad esempio, la stringa "distruggere". Essa è formata da 11 caratteri; il carattere in posizione 0 è d, il carattere in posizione 10 è l'ultima e, la lettera u si trova in posizione 5.

Il metodo `substring` con due argomenti interi, `x` e `y`, restituisce il riferimento a una stringa formata dai caratteri che vanno dalla posizione `x` fino alla posizione `y - 1`. In altre parole, il primo argomento specifica la posizione da cui cominciare a estrarre la sottostringa, mentre il secondo argomento specifica la prima posizione *dopo* la fine della sottostringa. Ad esempio:

```
"distruggere".substring(2, 9)
```

fornisce come risultato un riferimento alla stringa "strugge".

Il metodo `substring`, con un unico argomento di tipo `int`, restituisce un riferimento a una stringa formata da tutti i caratteri della stringa che si trovano tra la posizione specificata nell'argomento e la fine della stringa. Ad esempio:

```
"distruggere".substring(8)
```

restituisce un riferimento alla stringa "ere". Naturalmente, anziché utilizzare come argomenti alcune costanti, è possibile utilizzare variabili di tipo `int` o espressioni che forniscano risultati interi; invece di utilizzare un letterale stringa possiamo utilizzare una variabile contenente un riferimento a `String`. Ad esempio, in un programma in cui siano state effettuate le dichiarazioni:

```
String s1, s2;
int i, j;
```

si può scrivere l'istruzione:⁷

```
s2 = s1.substring(i, j);
```

⁷ Naturalmente è necessario che, prima dell'esecuzione dell'istruzione, alle variabili `s1`, `i` e `j` siano stati assegnati opportuni valori.

Un esempio riassuntivo

Giunti a questo punto, costruiamo un programma che utilizza alcuni metodi della classe `String` appena descritti. Compito di questo programma è quello di leggere una riga di testo, inserita dall'utente, e riprodurre la riga all'interno di una cornice formata da asterischi. Si suppone che il testo inserito dall'utente non superi i 74 caratteri (in questo modo si può costruire una cornice larga al massimo 80 caratteri, una dimensione tipica dei terminali alfanumerici). La larghezza della cornice dev'essere scelta in base alla lunghezza della frase inserita. Ad esempio, se l'utente inserisce la frase:

Ciao, come va?

il programma dovrà produrre il seguente output:

```
*****  
*      *  
* Ciao, come va? *  
*      *  
*****
```

La cornice è costituita da tre elementi fondamentali.

- Una riga centrale contenente la frase letta, preceduta da un asterisco e due spazi bianchi, e seguita da due spazi bianchi e un asterisco. In altre parole, questa riga, che chiameremo `rigaTesto`, è strutturata come:
`"* " + frase + " *"`.
- Una riga superiore (e una riga inferiore) costituita solo da asterischi. Chiameremo questa riga `rigaCornice`. Si noti che il numero di asterischi è uguale alla lunghezza della frase più 6.
- Una riga di mezzo, che chiameremo `rigaIntermedia`, che precede e segue la riga del testo. Questa riga inizia e finisce con un asterisco. Gli altri caratteri sono spazi. In particolare il numero degli spazi è uguale alla lunghezza della frase più 4.

A grandi linee possiamo utilizzare un algoritmo con la seguente struttura:

```
// fase di lettura  
leggi la frase  
  
// fase di calcolo  
costruisci rigaCornice  
costruisci rigaIntermedia  
costruisci rigaTesto  
  
// fase di scrittura  
scrivi rigaCornice
```

```
scrivi rigaIntermedia
scrivi rigaTesto
scrivi rigaIntermedia
scrivi rigaCornice
```

Come possiamo costruire `rigaCornice`? Avendo ipotizzato che il testo inserito dall'utente contenga tutt'alpiù 74 caratteri, la lunghezza di `rigaCornice` sarà al massimo di 80 caratteri. Dichiariamo all'interno del programma una variabile `rigaAsterischi` che si riferisce a una stringa formata da 80 asterischi. `rigaCornice` può essere costruita considerando una sottostringa di `rigaAsterischi`. Chiederemo quindi a `rigaAsterischi` di fornire, mediante il metodo `substring`, un riferimento a una sua sottostringa della lunghezza desiderata (ad esempio la sottostringa che inizia nella posizione 0 e termina nella posizione pari alla lunghezza della frase più 5). Possiamo procedere in maniera simile per `rigaIntermedia`. Per comodità introduciamo una variabile denominata `lunghezza`, in cui memorizziamo la lunghezza della frase letta (ottenibile mediante il metodo `length`). Ecco il codice completo della classe:

```
import prog.io.*;

class Cornice {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //le righe utilizzate per costruire la cornice
        String rigaAsterischi = "*****" +
                               "*****" +
                               "*****" +
                               "*****"; //in tutto
                                            //80 asterischi

        String rigaBianca =
                           " " +
                           " " +
                           " " +
                           " "; //in tutto 80 spazi

        String frase;
        int lunghezza;
        frase = in.readLine();
        lunghezza = frase.length();

        //costruzione delle righe che costituiranno la cornice
        String rigaCornice, rigaIntermedia, rigaTesto, spazi;
        rigaCornice = rigaAsterischi.substring(0, lunghezza + 6);
        spazi = rigaBianca.substring(0, lunghezza + 4);
```

```

rigaIntermedia = "*" + spazi + "*";
rigaTesto = "* " + frase + " *";

//visualizzazione della cornice
out.println(rigaCornice);
out.println(rigaIntermedia);
out.println(rigaTesto);
out.println(rigaIntermedia);
out.println(rigaCornice);
}

```

Esercizi

- 2.1 Riscrivete gli assegnamenti a `rigaIntermedia` e `rigaTesto` nella classe `Cornice` utilizzando il metodo `concat` in luogo dell'operatore `+` (ci sono almeno due soluzioni per ognuno degli assegnamenti).
- 2.2 L'output fornito dalla classe `Cornice` è allineato a sinistra sullo schermo. Modificate il metodo `main` in modo che l'output risulti centrato orizzontalmente (supponete di avere uno schermo di 80 colonne; la divisione tra interi è indicata dall'operatore `/`).
- 2.3 Scrivete una classe con un metodo `main` che, dopo avere richiesto all'utente in due interrogazioni successive di inserire il proprio cognome e nome, riporti in output il nome e il cognome, scritti sulla stessa riga utilizzando solo lettere maiuscole, all'interno di una cornice di asterischi.
- 2.4 Scrivete un programma Java che produca in output il valore della somma, della differenza, del prodotto, del quoziente e del resto della divisione di due numeri interi inseriti dall'utente (gli operatori per queste cinque operazioni si indicano, rispettivamente, con `+`, `-`, `*`, `/` e `%`). Il programma deve fornire ciascun risultato all'interno di un messaggio che indichi anche i dati e l'operazione applicata.
- 2.5 Se per il programma precedente si sono utilizzate più di tre variabili, riscrivetelo utilizzando al massimo tre variabili.

2.4 Variabili e tipi

Introduciamo ora le nozioni di variabile, tipo ed espressione, che sono fondamentali non solo nel linguaggio Java, ma in tutti i linguaggi di programmazione, e che approfondiremo nel prossimo capitolo.

Il linguaggio Java è un linguaggio *fortemente tipizzato*. Questo significa che il tipo di ogni variabile e di ogni espressione che appare in un programma può essere identificato leggendo il

testo del programma ed è noto pertanto *al momento della compilazione*. In particolare, durante la compilazione sono effettuati tutti i controlli relativi alla compatibilità dei tipi e sono stabilite eventuali conversioni implicite.

Consideriamo, ad esempio, l'espressione `x + y`. Il suo significato e le operazioni da eseguire per calcolarla dipendono dai tipi delle variabili `x` e `y`. Esaminiamo alcune possibilità.

- Se ambedue le variabili `x` e `y` sono state dichiarate di tipo `int`, l'operatore `+` denota una somma tra numeri interi; il risultato sarà di tipo `int`. Il calcolo viene fatto utilizzando i valori contenuti nelle due variabili e sommandoli, senza bisogno di effettuare alcuna trasformazione.
- Se ambedue le variabili `x` e `y` sono state dichiarate di tipo `String`, l'operatore `+` denota una concatenazione tra stringhe. Il risultato sarà una stringa o, più precisamente, un riferimento a un oggetto della classe `String`.
- Se la variabile `x` è stata dichiarata di tipo `String` e la variabile `y` di tipo `int`, l'operatore `+` denota una concatenazione tra stringhe. Anche in questo caso il risultato sarà una stringa. Poiché il valore contenuto nella variabile `y` non è una stringa, prima di calcolare il valore dell'espressione sarà necessario ottenere una stringa corrispondente a tale valore. In altre parole, in questo caso è necessaria un'operazione di conversione di tipo. Questa conversione è implicita: il programmatore non è tenuto a scrivere nulla per richiederla; le operazioni necessarie a ottenere la conversione sono automaticamente inserite dal compilatore nel codice generato sulla base dell'analisi dei tipi utilizzati nell'operazione.

Come vedremo più avanti, in talune circostanze è possibile richiedere esplicitamente una conversione di tipo.

Abbiamo evidenziato due differenti significati dell'operatore `+` (in realtà ce ne sono altri). Al fine di permettere al compilatore (ma anche a chi legge il testo del programma) di attribuire a un operatore il significato corretto, e quindi di decidere quali azioni andranno attuate in corrispondenza di questo durante l'esecuzione, è obbligatorio *dichiarare* le variabili prima di utilizzarle, specificandone il tipo. La stessa variabile non può essere poi dichiarata di nuovo di un tipo differente nella stessa porzione di codice (eccetto che nei casi di *adombramento* che esamineremo più avanti).

Abbiamo già visto alcuni esempi di dichiarazioni di variabili: dopo avere scritto il nome del tipo, si scrivono gli identificatori delle variabili che si desidera dichiarare separandoli con una virgola. Ad esempio:

```
int x, y;
String nome;
```

Insieme con la dichiarazione è possibile assegnare alle variabili un valore iniziale; in questo caso (dichiarazione più assegnamento) partiamo di *definizione* di variabile. Ad esempio, le istruzioni:

```
int i = 4, j = 3;
String nome = "pippo";
```

dichiarano le variabili `i` e `j` assegnando loro, rispettivamente, i valori 4 e 3, e dichiarano la variabile `nome` assegnando a essa un riferimento alla stringa "pippo".

Abbiamo già osservato che le variabili di tipo `int` contengono direttamente un valore intero, mentre le variabili il cui tipo sia il nome di una classe, come `String`, non contengono direttamente l'oggetto, ma solo un riferimento a esso. Più in generale, in Java i tipi delle variabili si dividono in due categorie.

- *Tipi primitivi* (come `int`).

Una variabile di tipo primitivo può essere immaginata come una scatola in grado di contenere direttamente un valore di tipo primitivo.

- *Tipi riferimento* (come `String`).

Una variabile di un tipo riferimento non contiene direttamente un valore, ma contiene un'informazione, il *riferimento*, che permette di accedere al valore o, più precisamente, all'oggetto riferito.

Mentre i tipi primitivi di Java sono un insieme fissato (in tutto sono 8), i tipi riferimento corrispondono alle classi, come `String`, `ConsoleOutputManager` e `ConsoleInputManager` che abbiamo già introdotto, e alle *interfacce* e agli *array* che introduciamo più avanti; i tipi riferimento sono virtualmente infiniti in quanto, come vedremo nella seconda parte di questo libro, il programmatore è libero di definire nuove classi, interfacce e array.

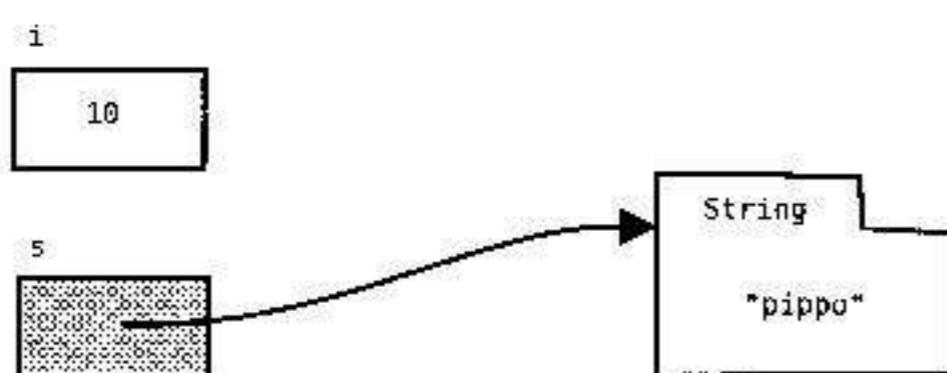


Figura 2.4 Tipi primitivi e tipi riferimento.

La differenza fra le due categorie di variabili è illustrata nella Figura 2.4, in cui è rappresentata la situazione in memoria relativa alle seguenti definizioni:

```
int i = 10;
String s = "pippo";
```

Come vedremo nel Capitolo 7, oltre alle classi disponibili nelle librerie di Java, il programmatore può definire nuove classi e utilizzarle per dichiarare variabili di tipo riferimento. Pertanto non c'è limite ai possibili tipi riferimento che si possono utilizzare nei programmi Java.

Come abbiamo già visto, per dichiarare una variabile di tipo riferimento a una certa classe, è sufficiente scrivere il nome della classe prima del nome della variabile. Ad esempio scrivendo:

```
String s;
```

dichiariamo che `s` è una variabile in grado di contenere un riferimento a un oggetto della classe `String`. Una variabile di tipo riferimento potrebbe anche non riferirsi ad alcun oggetto. A tale scopo si utilizza un particolare valore, indicato dal letterale `null`. Ad esempio, possiamo scrivere, l'assegnamento:

```
s = null;
```

Il tentativo di accedere a un oggetto tramite un riferimento `null` provoca un errore di esecuzione. Ad esempio, se dopo l'istruzione precedente tentassimo di eseguire

```
int x = s.length();
```

si avrebbe un errore: verrebbe infatti richiesto un servizio (l'esecuzione del metodo `length`) a un oggetto inesistente.

Per programmare in Java è fondamentale comprendere la differenza tra tipi primitivi e riferimenti. In particolare occorre ricordare sempre che una variabile di un tipo riferimento *non contiene* l'oggetto, ma contiene solo un riferimento all'oggetto. Consideriamo, ad esempio, un semplice assegnamento della forma

```
v = u;
```

dove le variabili `u` e `v` siano state dichiarate dello stesso tipo. Indipendentemente dal tipo di `u` e `v`, l'assegnamento è eseguito copiando in `v` il contenuto di `u`. Se `u` è di un tipo primitivo, come `int`, l'effetto è quello di porre in `v` lo stesso valore di tipo `int` che, prima dell'assegnamento, si trovava in `u`. In altre parole viene creata una copia del valore di tipo `int` presente in `u`, che viene posta in `v`. Ad esempio, se il valore di `u` prima dell'assegnamento è 10, la situazione in memoria prima e dopo l'assegnamento è rappresentata nella Figura 2.5. Si osservi che il valore contenuto in `v` prima dell'assegnamento sarà perso.

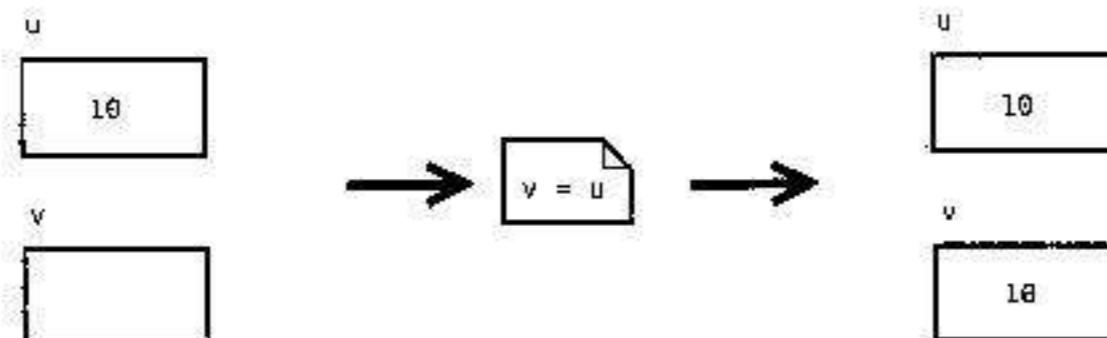


Figura 2.5 Assegnamento fra variabili di tipo primitivo.

Se invece `u` e `v` sono di un tipo riferimento, come ad esempio `String`, il contenuto di `u` è il riferimento a una stringa, non la stringa stessa, e in `v` sarà copiato tale riferimento. In altre parole, dopo l'assegnamento, `u` e `v` saranno due riferimenti al medesimo oggetto di tipo `String`. Ad esempio, se `u` fa riferimento all'oggetto che rappresenta la stringa "pippo", la situazione in memoria prima e dopo l'assegnamento è rappresentata nella Figura 2.6.

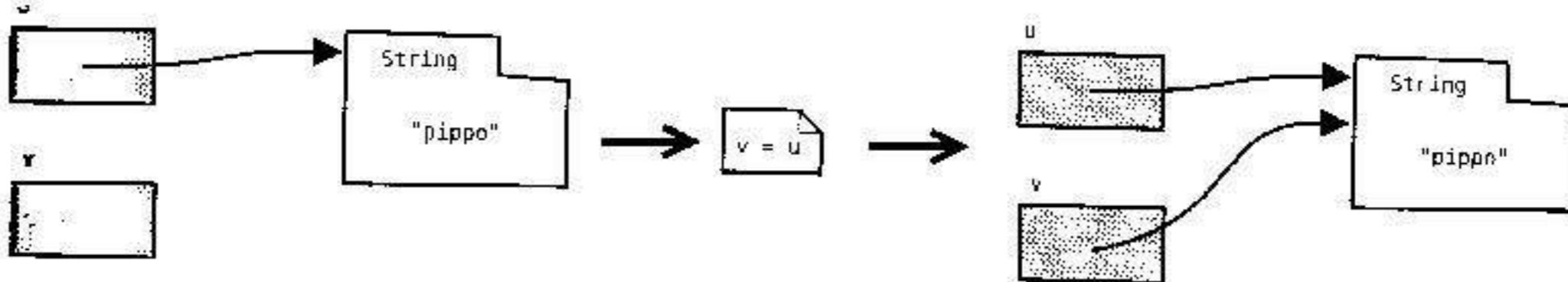


Figura 2.6 Assegnamento fra variabili di tipo riferimento.

Esercizi riassuntivi

Gli esercizi presentati qui richiedono la costruzione di semplici applicazioni utilizzando classi disponibili nelle librerie del testo. Per la soluzione di ciascun esercizio si suggerisce di adottare questo procedimento.

- Leggete attentamente la descrizione delle classi fornite.
- Leggete quali sono i compiti che l'applicazione richiesta deve svolgere, osservando con attenzione gli eventuali esempi forniti.
- Scrivete la sequenza dei passi fondamentali dell'algoritmo risolutivo, come è stato fatto nell'esempio dell'applicazione *Cornice*.
- Analizzate singolarmente i passi individuati: alcuni richiederanno la creazione di nuovi oggetti, altri richiederanno invece lo svolgimento di determinati compiti. Per ciascun compito chiedetevi quale oggetto tra quelli costruiti dovrà occuparsene e in che modo, cioè tramite quale metodo. A questo scopo è utile rifarsi sempre alla documentazione delle classi.
- Codificate l'applicazione in linguaggio Java. Ricordatevi di importare le classi necessarie.
- Infine verificate quanto fatto scrivendo l'applicazione con il vostro editor, compilandola e mandandola in esecuzione.
- Nel caso l'applicazione richieda lo svolgimento di diversi compiti, si consiglia di cominciare mettendo a punto una prima versione in cui se ne svolge uno solo; gli altri compiti si aggiungeranno man mano in versioni successive.

2.6 Si consideri la classe `Frazione` del package `prog.util` fornito con il testo, i cui oggetti rappresentano frazioni. Questa classe mette a disposizione il costruttore:

- `public Frazione(int x, int y)`
Costruisce una nuova frazione il cui valore è il rapporto fra il primo argomento e il secondo argomento.

Inoltre mette a disposizione, fra gli altri, il metodo:

- `public String toString()`

Restituisce una stringa di caratteri che descrive la frazione rappresentata dall'oggetto che esegue il metodo (la stringa è del tipo "4/3", oppure "4" quando la frazione ha denominatore uguale a 1).

Scrivete un'applicazione che, lette da tastiera due stringhe, visualizzi la lunghezza delle due stringhe e quindi la media delle loro lunghezze espressa come frazione. Due esempi di esecuzione sono i seguenti:

```
Inserisci la prima stringa: pippo
Inserisci la seconda stringa: topolino
La lunghezza di pippo è: 5
La lunghezza di topolino è: 8
La media delle loro lunghezze è 13/2
```

```
Inserisci la prima stringa: pippo
Inserisci la seconda stringa: pluto
La lunghezza di pippo è: 5
La lunghezza di pluto è: 5
La media delle loro lunghezze è 5
```

2.7 Il package `prog.utili` fornisce una classe `Intero` le cui istanze rappresentano numeri interi. Tale classe mette a disposizione il seguente costruttore:

- `public Intero(int x)`

Costruisce un nuovo oggetto che rappresenta il numero intero fornito tramite l'argomento.

Fra gli altri la classe mette a disposizione questi metodi:

- `public int intValue()`

Restituisce il valore intero rappresentato dall'oggetto che esegue il metodo.

- `public String toString()`

Costruisce una stringa contenente il numero rappresentato dall'oggetto che esegue il metodo, espresso in lettere; ad esempio, se l'oggetto che esegue il metodo rappresenta il numero 34, il metodo restituirà la stringa "trentaquattro".

Scrivete un'applicazione che legga due interi e ne calcoli la somma. L'applicazione dovrà scrivere l'operazione effettuata, esprimendo i numeri prima in cifre e poi in lettere (per esprimere i numeri in lettere servirsi di quanto offerto dalla classe `Intero`). Ad esempio un'esecuzione dell'applicazione potrebbe essere la seguente:

```

primo addendo? 4
secondo addendo? 5
4 + 5 = 9
quattro + cinque = nove

```

- 2.8 Scrivete un'applicazione che, letta una stringa, comunichi all'utente la sua lunghezza, espressa in cifre e in lettere. L'applicazione dovrà poi comunicare all'utente sia in cifre che in lettere, la lunghezza della stringa che esprime in lettere la lunghezza della stringa letta. Infine l'applicazione dovrà comunicare sia in cifre che in lettere, la lunghezza della stringa che esprime in cifre la lunghezza della stringa letta. Ad esempio un'esecuzione dell'applicazione potrebbe essere la seguente:

```

Stringa? Pippo
La lunghezza della stringa "Pippo" e' 5 (cinque)
La lunghezza della stringa "cinque" e' 6 (sei)
La lunghezza della stringa "5" e' 1 (uno)

```

Nota: per utilizzare le virgolette in una stringa, cioè per togliere la loro funzione di delimitatori, si utilizza la sequenza di escape \".

- 2.9 Il package `prog.utili` fornisce una classe `Importo` i cui oggetti rappresentano importi in Euro. Tale classe mette a disposizione questi costruttori:

- `public Importo(int x)`
Costruisce un nuovo oggetto che rappresenta un importo in cui il numero di Euro, senza centesimi, è specificato dall'argomento.
- `public Importo(int x, int y)`
Costruisce un nuovo oggetto che rappresenta un importo in cui il numero di Euro è specificato dal primo argomento e il numero di centesimi di Euro dal secondo argomento.

Fra gli altri, la classe mette a disposizione questi metodi:

- `public int getEuro()`
Restituisce il numero di Euro (senza centesimi) dell'importo rappresentato dall'oggetto che esegue il metodo.
- `public int getCent()`
Restituisce il numero di centesimi dell'importo rappresentato dall'oggetto che esegue il metodo.
- `public Importo piu(Importo x)`
Restituisce il riferimento a un nuovo oggetto che rappresenta l'importo ottenuto sommando l'importo fornito tramite l'argomento a quello rappresentato dall'oggetto che esegue il metodo.

- **public Importo meno(Importo x)**
Restituisce il riferimento a un nuovo oggetto che rappresenta l'importo ottenuto sottraendo l'importo fornito tramite l'argomento a quello rappresentato dall'oggetto che esegue il metodo.
- **public Importo per(int x)**
Restituisce il riferimento a un nuovo oggetto che rappresenta l'importo ottenuto moltiplicando l'importo rappresentato dall'oggetto che esegue il metodo per l'intero fornito tramite l'argomento.
- **public Importo diviso(int x)**
Restituisce il riferimento a un nuovo oggetto che rappresenta l'importo ottenuto dividendo l'importo rappresentato dall'oggetto che esegue il metodo per l'intero fornito tramite l'argomento.
- **public int toLire()**
Restituisce il valore in Lire corrispondente all'importo rappresentato dall'oggetto che esegue il metodo.
- **public String toString()**
Restituisce il riferimento a una stringa che rappresenta il valore dell'importo che esegue il metodo (la stringa contiene nelle due posizioni più a destra le cifre corrispondenti ai centesimi, anche se queste sono zero).

Scrivete un'applicazione che legga un prezzo espresso in Euro (parte intera e centesimi) e una percentuale di sconto (come numero intero) e comunichi:

- l'importo letto in Euro;
- lo sconto in Euro;
- l'importo scontato in Euro e in Lire.

Esempio di esecuzione:

```
Prezzo: euro? 10
       cent? 60
Sconto percentuale? 5
Prezzo: EURO 10,60
Sconto: EURO 0,53
Prezzo scontato: EURO 10,07 (Lire 19498)
```

2.10 Modificate l'output dell'applicazione sviluppata per l'esercizio precedente, in modo che visualizzi l'output allineato come in uno scontrino. Esempio di esecuzione:

```
Prezzo: euro? 10
       cent? 60
Sconto percentuale? 5
```

IMPORTO	10,60
SCONTO 5%	0,53
DA PAGARE	EURO 10,07
	(Lire 19498)

Basi del linguaggio

Nel capitolo precedente abbiamo introdotto alcuni elementi del linguaggio Java, già sufficienti a scrivere applicazioni che utilizzino classi e oggetti. In questo capitolo approfondiremo lo studio degli elementi di base del linguaggio Java. In particolare, dopo aver introdotto la classe `Frazione` che utilizzeremo ampiamente negli esempi, tratteremo in dettaglio le strutture di controllo che costituiscono l'implementazione Java delle strutture presentate nel Paragrafo 1.9. Vedremo che i tipi primitivi (la cui trattazione proseguirà nel capitolo successivo) e le strutture di controllo di Java presentano molte somiglianze con gli analoghi di altri linguaggi di programmazione moderni come Pascal e C; si tratta infatti di aspetti per nulla legati alla natura *Object Oriented* del linguaggio. In particolare, Java ha ereditato la sintassi e la semantica delle strutture di controllo dal linguaggio C.

3.1 La classe Frazione

In questo paragrafo presentiamo la classe `Frazione` che utilizzeremo negli esempi di questo capitolo.¹

Gli oggetti di questa classe modellano frazioni: una frazione sarà quindi un oggetto in grado di compiere operazioni. Iniziamo a vedere quali sono le operazioni che ci aspettiamo che un oggetto `Frazione` possa eseguire.

- *Operazioni aritmetiche.*

Nella classe `String` possiamo chiedere a una stringa di calcolare la propria concatenazione con un'altra stringa, producendo così una nuova stringa (metodo `concat`). Facciamo lo stesso per le frazioni. In pratica vogliamo disporre di quattro metodi: `piu`, `meno`, `per`, `diviso`, corrispondenti alle operazioni aritmetiche elementari, che ricevano come argomento una frazione e restituiscano come risultato una nuova frazione, il cui valore è ottenuto applicando l'operazione alla frazione rappresentata dall'oggetto che esegue il metodo e alla frazione fornita tramite l'argomento. Ad esempio, se `a` e `b` sono frazioni, la chiamata

¹ Il bytecode della classe è fornito tra le classi sviluppate per il testo nel package `prog.utili`. Pertanto, quando si scrive codice che utilizza la classe, occorrerà ricordarsi di inserire l'opportuna direttiva d'importazione.

`a.meno(b)` dovrà restituire il riferimento a una nuova frazione, il cui valore è uguale a quello contenuto in `a` meno quello contenuto in `b`.

Ecco i prototipi dei metodi indicati sopra:

- `Frazione piu(Frazione f)`
- `Frazione meno(Frazione f)`
- `Frazione per(Frazione f)`
- `Frazione diviso(Frazione f)`

- *Operazioni di confronto.*

Servono per confrontare la frazione rappresentata dall'oggetto con un'altra frazione. Introduciamo tre metodi: per verificare l'uguaglianza, per verificare se la frazione è minore dell'altra e per verificare se è maggiore. Questi metodi possono fornire come risposta uno dei due valori di verità, *vero* o *falso*, rappresentati in Java dalle costanti `true` e `false` del tipo primitivo `boolean`. Pertanto i prototipi dei metodi saranno:

- `boolean equals(Frazione f)`
- `boolean isMinore(Frazione f)`²
- `boolean isMaggiore(Frazione f)`

- *Rappresentazione.*

È utile disporre di un metodo che restituisca una stringa che rappresenti il contenuto dell'oggetto. Tutte le classi forniscono un metodo di questo genere, che prende il nome di `toString`. Il prototipo sarà:

- `String toString()`

Il metodo `toString`, nel nostro caso applicato ad esempio a un oggetto che rappresenta la frazione $\frac{2}{3}$, produce la stringa "2/3".

Forniamo ora una breve descrizione della classe che riassume e completa quanto detto sopra. La classe `Frazione` ha due costruttori:

- `public Frazione(int x)`
Costruisce una nuova `Frazione` il cui numeratore è uguale all'argomento e il cui denominatore è 1.
- `public Frazione(int x, int y)`
Costruisce una nuova `Frazione` il cui valore è il rapporto fra il primo argomento e il secondo argomento.

² Come osserveremo in seguito, è comune utilizzare come nomi dei metodi verbi o frasi. In molti casi il nome più corretto per metodi che restituiscono un `boolean` richiederebbe l'uso della terza persona del verbo essere (ad esempio nel caso del metodo `inCircolazione` il nome del metodo sarebbe `èMinore`). Per evitare di utilizzare lettere accentate nei nomi dei metodi, cosa fattibile ma poco pratica, abbiamo deciso di sostituire la terza persona del verbo essere con la forma inglese.

Dis

Quest
gram
zione

Dispone inoltre dei seguenti metodi:

- **public Frazione piu(Frazione f)**
Restituisce il riferimento a un nuovo oggetto che rappresenta la frazione ottenuta sommando la frazione specificata come argomento a quella che esegue il metodo.
- **public Frazione meno(Frazione f)**
Restituisce il riferimento a un nuovo oggetto che rappresenta la frazione ottenuta sottraendo la frazione specificata come argomento da quella che esegue il metodo.
- **public Frazione per(Frazione f)**
Restituisce il riferimento a un nuovo oggetto che rappresenta la frazione ottenuta moltiplicando la frazione specificata come argomento per quella che esegue il metodo.
- **public Frazione diviso(Frazione f)**
Restituisce il riferimento a un nuovo oggetto che rappresenta la frazione ottenuta dividendo la frazione che esegue il metodo per quella specificata come argomento.
- **public boolean equals(Frazione f)**
Confronta la frazione rappresentata dall'oggetto che esegue il metodo con la frazione specificata come argomento. Restituisce `true` se le due frazioni hanno lo stesso valore, `false` in caso contrario.
- **public boolean isMinore(Frazione f)**
Confronta la frazione rappresentata dall'oggetto che esegue il metodo con la frazione specificata come argomento. Restituisce `true` se la frazione che esegue il metodo è minore di quella specificata come argomento, `false` in caso contrario.
- **public boolean isMaggiore(Frazione f)**
Confronta la frazione rappresentata dall'oggetto che esegue il metodo con la frazione specificata come argomento. Restituisce `true` se la frazione che esegue il metodo è maggiore di quella specificata come argomento, `false` in caso contrario.
- **public int getNumeratore()**
Restituisce il numeratore della frazione rappresentata dall'oggetto che esegue il metodo.
- **public int getDenominatore()**
Restituisce il denominatore della frazione rappresentata dall'oggetto che esegue il metodo.
- **public String toString()**
Restituisce una stringa di caratteri che descrive la frazione rappresentata dall'oggetto che esegue il metodo.

Questo tipo di presentazione delle classi è simile a quella fornita dalle *API* (*Application Programmer Interface*, interfaccia di programmazione applicativa) che contengono la documentazione relativa alle classi fornite insieme con Java. Rispetto a quanto visto finora, osserviamo che

nel prototipo dei metodi e dei costruttori compare la parola chiave `public`, che è un modificatore di visibilità. Tratteremo in dettaglio i vari modificatori nel Capitolo 7. Per ora ci basta sapere che i metodi e i modificatori dichiarati `public` possono essere utilizzati liberamente dagli utenti della classe.

Come primo esempio d'uso della classe `Frazione` scriviamo un'applicazione che, lette due frazioni, ne calcola e visualizza la somma. Dato che non abbiamo metodi che ci consentano di leggere direttamente una frazione, dovremo ricorrere ai metodi di lettura di cui disponiamo. Poiché la classe `Frazione` modella frazioni e dispone di un costruttore che prende numeratore e denominatore come argomenti, possiamo ricorrere al metodo `readInt` della classe `ConsoleInputManager` per leggere il numeratore e il denominatore della frazione, e quindi utilizzarli per costruire la frazione. Se `in` è una variabile di tipo `ConsoleInputManager`, possiamo leggere la prima frazione nel modo seguente:

```
int num, den;

num = in.readInt("Numeratore prima frazione> ");
den = in.readInt("Denominatore prima frazione> ");
Frazione f1 = new Frazione(num, den);
```

In modo analogo possiamo leggere la seconda frazione salvandone il riferimento nella variabile `f2`. A questo punto, per calcolare la somma delle due frazioni possiamo utilizzare il metodo `piu` di una delle due frazioni, ad esempio `f1`, e fornire l'altra come argomento scrivendo:

```
f1.piu(f2)
```

Dato che questo metodo restituisce un riferimento a un oggetto di tipo `Frazione`, per poter accedere successivamente a questo nuovo oggetto dovremo salvarne il riferimento in un'opportuna variabile. Il codice dell'intera applicazione è il seguente.

```
import prog.io.*;
import prog.utili.Frazione;

class SommaFrazioni {

    static public void main (String[] args){
        //predisposizione dei canali di comunicazione
        ConsoleOutputManager out = new ConsoleOutputManager();
        ConsoleInputManager in = new ConsoleInputManager();

        int num, den;

        //lettura e costruzione della prima frazione
        num = in.readInt("Numeratore prima frazione> ");
        den = in.readInt("Denominatore prima frazione> ");
```

}

Ese

3.1

3.2

3.3

3.2

Scri
scrive
costr
segue

```

Frazione f1 = new Frazione(num, den);

//lettura e costruzione della seconda frazione
num = in.readInt("Numeratore seconda frazione> ");
den = in.readInt("Denominatore seconda frazione> ");
Frazione f2 = new Frazione(num, den);

Frazione somma = f1.piu(f2);

//comunicazione
out.println("La somma delle frazioni " + f1.toString() +
            " e " + f2.toString() + " è " + somma.toString());
}
}

```

Esercizi

- 3.1 Scrivete un'applicazione che legga due frazioni e visualizzi la loro somma, la loro differenza, il loro prodotto e il quoziente della prima frazione per la seconda.
- 3.2 Scrivete un'applicazione che legga da tastiera una frazione e scriva a video il suo quadrato.
- 3.3 Scrivete un'applicazione che legga due frazioni, calcoli una nuova frazione, il cui valore sia la media delle frazioni lette, comunichi il risultato prima in cifre (ad esempio "9/16") e poi in lettere, indicando il numeratore in lettere minuscole e il denominatore in maiuscole (ad esempio "nove/SEDICI"). Servitevi, oltre che della classe `Frazione`, anche della classe `Intero` del package `prog.utili`.

3.2 L'istruzione if-else

Scriviamo ora una seconda applicazione che utilizza la classe `Frazione`. Supponiamo di dover scrivere un programma che, lette due frazioni, dica se sono uguali oppure no. Utilizzando il costrutto di selezione che abbiamo introdotto nel Paragrafo 1.9 possiamo procedere nel modo seguente:

```

leggi il numeratore e il denominatore della prima frazione e costruisci
l'oggetto corrispondente
leggi il numeratore e il denominatore della seconda frazione e costruisci
l'oggetto corrispondente
SE le due frazioni sono uguali
    ALLORA
        stampa a video "Le due frazioni sono uguali"

```

ALTRIMENTI

stampa a video "Le due frazioni sono diverse"

FINESE

L'unico elemento del linguaggio che ci manca per essere in grado di codificare questo programma è il costrutto di selezione. In Java tale costrutto è realizzato dall'istruzione **if-else**, che ha questa forma:

```
if (condizione)
    istruzione1
else
    istruzione2
```

Dove:

- *condizione* è una qualunque espressione che restituisce un valore di tipo **boolean**, scritta obbligatoriamente tra parentesi tonde
- *istruzione1* e *istruzione2* sono istruzioni singole oppure *blocchi di istruzioni*, cioè sequenze di istruzioni racchiuse tra parentesi graffe.

Generalmente si parla di *ramo then* di un'istruzione **if-else** in riferimento all'istruzione o al blocco di istruzioni costituite da *istruzione1*; si parla invece di *ramo else* in riferimento all'istruzione o al blocco di istruzioni costituite da *istruzione2*.

L'esecuzione di un'istruzione **if-else** avviene come segue.

- (1) Per prima cosa viene valutata la condizione; tale valutazione restituisce uno dei due valori: **true** o **false**.
- (2) A questo punto l'esecuzione prosegue in modo diverso a seconda che il valore della condizione sia **true** o **false**:
 - se la condizione risulta vera (cioè se il suo valore è **true**), viene eseguita *istruzione1*;
 - se invece la condizione è falsa (cioè se il suo valore è **false**), viene eseguita *istruzione2*.
- (3) L'esecuzione riprende in ogni caso dalla prima istruzione che segue l'istruzione **if-else**.

Esiste anche una forma semplificata dell'istruzione di selezione senza il ramo **else**, che è chiamata semplicemente **istruzione if**, il cui schema è:

```
if (condizione)
    istruzione1
```

In questo caso, se in fase di esecuzione la condizione è vera, viene eseguita l'istruzione (o il blocco di istruzioni) *istruzione1*, dopodiché l'esecuzione riprende dalla prima istruzione che segue il costrutto **if**. Se invece la condizione è falsa, l'esecuzione prosegue direttamente dalla prima istruzione che segue il costrutto **if**.

Tornando all'applicazione che vogliamo realizzare, dovremo utilizzare l'istruzione **if-else** con una condizione che permetta di verificare se le due frazioni fornite dall'utente sono uguali oppure diverse. Per far questo possiamo utilizzare il metodo `equals` della classe `Frazione`. Esso riceve come argomento una frazione e restituisce un valore di tipo `boolean`. Come specifica il suo contratto, restituisce `true` se la frazione fornita come argomento ha lo stesso valore di quella che lo esegue, e `false` in caso contrario. Quindi, se `f1` e `f2` sono riferimenti a oggetti di tipo `Frazione`, la condizione che ci serve è `f1.equals(f2)`. Il programma completo è il seguente:

```
import prog.io.*;
import prog.utili.Frazione;

class ConfrontoFrazioni {

    static public void main (String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleOutputManager out = new ConsoleOutputManager();
        ConsoleInputManager in = new ConsoleInputManager();

        int num, den;

        //lettura e costruzione della prima frazione
        num = in.readInt("Numeratore prima frazione> ");
        den = in.readInt("Denominatore prima frazione> ");
        Frazione f1 = new Frazione(num, den);

        //lettura e costruzione della seconda frazione
        num = in.readInt("Numeratore seconda frazione> ");
        den = in.readInt("Denominatore seconda frazione> ");
        Frazione f2 = new Frazione(num, den);

        //confronto ...
        if (f1.equals(f2))
            out.println("Le due frazioni sono uguali");
        else
            out.println("Le due frazioni sono diverse");
    }
}
```

Descrivendo nel primo capitolo le strutture di controllo fondamentali abbiamo visto come sia possibile innestarle tra loro. Naturalmente ciò è possibile anche nel linguaggio Java. In particolare è possibile innestare un'istruzione `if` in un'altra istruzione `if`. Ad esempio, se abbiamo dichiarato le variabili `x`, `y` e `z` di tipo `int`, potremmo scrivere:

```

if (x == 1)
    if (y == 1)
        z = x + y;
    else
        z = x * y;
else
    z = x - y;

```

Osserviamo che possono verificarsi i seguenti casi.

- Le variabili `x` e `y` contengono inizialmente il valore 1:
viene eseguita la somma, assegnando così a `z` il valore 2.
- La variabile `x` contiene inizialmente 1, mentre `y` contiene un valore diverso da 1:
viene eseguito il primo ramo `else`, che assegna a `z` il prodotto dei valori delle due variabili.
- La variabile `x` contiene un valore diverso da 1:
a prescindere dal valore contenuto nella variabile `y`, sarà sempre eseguito il secondo ramo `else`, assegnando a `z` la differenza tra i valori delle due variabili.

Ricordiamo, inoltre, che l'indentazione utilizzata scrivendo il codice dei programmi è di fondamentale importanza per chi legge, ma non ha alcun significato per il compilatore. Per il compilatore il codice precedente equivale a

```

if (x == 1)
    if (y == 1)
        z = x + y;
    else
        z = x * y;
    else
        z = x - y;

```

e anche a

```
if (x == 1) if (y == 1) z = x + y; else z = x * y; else z = x - y;
```

Come abbiamo visto, in un'istruzione `if` può essere omessa la parte `else`. Analizziamo ora la situazione in cui ci sono due `if` innestati con un unico `else`.

Consideriamo il seguente esempio, in cui abbiamo scritto appositamente sulla stessa riga le due istruzioni `if` innestate:

```

import prog.io.*;
class Mistero {
    public static void main(String[] args) {

```

```

//predisposizione dei canali di comunicazione
ConsoleInputManager in = new ConsoleInputManager();
ConsoleOutputManager out = new ConsoleOutputManager();

int x = in.readInt("Inserire il primo numero ");
int y = in.readInt("Inserire il secondo numero ");
int z = 100;

if (x == 1) if (y == 1) z = x + y; else z = x - y;
out.println("Il risultato è " + z);
}
}

```

Dal punto di vista del compilatore, la riga contenente le due istruzioni `if` è equivalente al seguente frammento:

```

if (x == 1)
    if (y == 1)
        z = x + y;
    else
        z = x - y;

```

e anche al seguente:

```

if (x == 1)
    if (y == 1)
        z = x + y;
else
    z = x - y;

```

La prima indentazione suggerisce che l'`else` appartiene all'istruzione `if` più interna, mentre la seconda suggerisce che l'`else` si riferisce all'istruzione `if` più esterna. Indipendentemente da come è inserito il testo, il compilatore utilizza la seguente regola:

- in una sequenza di `if` innestati, un `else` è sempre associato al primo `if` che lo precede, per il quale non sia stato ancora individuato un `else`.

In questo caso l'`else` verrà dunque associato all'`if` più interno. Pertanto, se le variabili `x` e `y` contengono rispettivamente i valori 1 e 0, sarà eseguito l'`else`, assegnando così a `z` la differenza, cioè 1.

Per associare l'unico `else` al primo `if` si devono utilizzare le parentesi graffe:

```

if (x == 1) {
    if (y == 1)
        z = x + y;
} else
    z = x - y;

```

In questo caso, se le variabili `x` e `y` contengono rispettivamente i valori `1` e `0`, non viene eseguita alcuna istruzione.

Esercizi

3.4 Scrivete un'applicazione che legga da tastiera due frazioni e le riscriva a video in ordine crescente.

3.5 Scrivete un'applicazione che, dopo avere richiesto all'utente in due interrogazioni successive di inserire il proprio cognome e nome, riporti in output il nome e il cognome su due righe differenti, all'interno di una cornice di asterischi. La larghezza della cornice dovrà essere dimensionata in base alla più lunga delle due stringhe contenenti il nome e il cognome. Esempi:

```
*****
*          *
* Giuseppe *
*          *
*   Verdi  *
*          *
*****
```

```
*****
*          *
* Dante   *
*          *
* Alighieri *
*          *
*****
```

3.6 Siano `x`, `y` e `z` tre variabili di tipo `int`. Considerate il seguente frammento di codice:

```
if (x != 0) if (y == 1) if (x == y) z = x + y; else z = x * y;
else z = x - y;
```

- (1) Indicate a quali `if` si riferiscono i due `else`.
- (2) Riscrivete la riga di codice utilizzando l'indentazione corretta.
- (3) Per ognuno degli assegnamenti che compaiono nella riga di codice precedente, indicate dei valori iniziali delle variabili `x` e `y` affinché l'assegnamento sia eseguito.

3.7 La classe `String` mette a disposizione un metodo `compareTo` che riceve come argomento un riferimento a un oggetto di tipo `String` e restituisce un valore di tipo `int`. In particolare il metodo restituisce:

- zero se la stringa fornita come argomento è uguale a quella che esegue il metodo;
- un valore minore di zero se la stringa che esegue il metodo è lessicograficamente minore di quella fornita come argomento, cioè se la precede nell'ordine alfabetico;
- un valore maggiore di zero se la stringa che esegue il metodo è lessicograficamente maggiore di quella fornita come argomento, cioè se la segue nell'ordine alfabetico.

Utilizzando tale metodo scrivete un'applicazione che, lette due stringhe inserite dall'utente indichi se sono uguali o diverse. Nel caso le stringhe siano diverse l'applicazione dovrà visualizzarle prima in ordine alfabetico e poi in ordine di lunghezza. Alcuni esempi di esecuzione sono i seguenti.

Prima stringa? cane

Seconda stringa? bovino

Ordine alfabetico:

bovino

cane

Ordine di lunghezza:

cane

bovino

Prima stringa? cane

Seconda stringa? cane

Le due stringhe sono uguali

Prima stringa? cane

Seconda stringa? topo

Ordine alfabetico:

cane

topo

Le due stringhe hanno la stessa lunghezza

- 3.8 La lunghezza della stringa di caratteri che esprime il numero 12 nella lingua italiana è 6, pari cioè alla metà del numero stesso. Altri numeri, come 24 e 6 hanno questa curiosa proprietà. Utilizzando la classe `Intero` del package `prog.utili` descritta nell'Esercizio 2.7, scrivete un'applicazione che legga un numero e verifichi se gode di questa proprietà. Due esempi di esecuzione dell'applicazione sono i seguenti:

numero? 12

La lunghezza della parola dodici e' 6, uguale alla metà di 12

numero? 20

La lunghezza della parola venti e' 5, mentre la metà di 20 e' 6

3.3 Il tipo primitivo boolean

Come evidenziato dal prototipo, i metodi `equals`, `isMaggiore` e `isMinore` della classe `Frazione` restituiscono un valore di tipo `boolean`. Esso è un tipo primitivo di Java che ha un ruolo fondamentale nelle strutture di controllo. Il tipo `boolean` ha solo due valori possibili, denotati dalle costanti (letterali del linguaggio) `false` e `true`, utilizzate per indicare, rispettivamente, i valori di verità *falso* e *vero*.

Le espressioni booleane, cioè le espressioni che restituiscono un valore di tipo `boolean`, sono dette *condizioni*. Le più semplici sono quelle ottenute confrontando espressioni di tipo primitivo mediante uno degli *operatori relazionali*, che sono:

- > maggiore di
- >= maggiore o uguale a
- < minore di
- <= minore o uguale a
- == uguale a
- != diverso da

Ad esempio, se `x` e `y` sono due variabili di tipo `int`, l'espressione `x == y` restituisce `true` se `x` e `y` contengono il medesimo valore, e `false` in caso contrario. L'espressione `x <= y` restituisce `true` se il valore contenuto in `x` è minore o uguale a quello contenuto in `y`, e `false` in caso contrario.

Si osservi che le espressioni costituite da una sola variabile o da una sola costante sono le più semplici che si possono scrivere in Java. Come vedremo in seguito, le espressioni possono essere costituite da sequenze ben più complesse di operatori e operandi. Ad esempio una semplice espressione intera che contiene un operatore è `y + 5`; l'espressione booleana `x == y + 5` restituisce `true` se `x` contiene il medesimo valore dell'espressione `y + 5`, cioè il valore ottenuto sommando 5 al valore contenuto in `y`.

Gli operatori `==` e `!=` possono essere utilizzati anche per confrontare espressioni il cui risultato è il riferimento a un oggetto. Poiché questi operatori confrontano i risultati delle espressioni, ciò che sarà confrontato sono i riferimenti e non gli oggetti, come si potrebbe essere indotti a credere. Ad esempio, se `u` e `v` sono variabili riferimento di tipo `String`, è possibile scrivere l'espressione `u == v`; tale espressione risulta vera solo se `u` e `v` fanno riferimento allo stesso oggetto, come nel caso rappresentato nella Figura 3.1(a). Se invece `u` e `v` fanno riferimento a oggetti distinti, come nel caso descritto nella Figura 3.1(b), allora l'espressione `u == v` risulta falsa anche se i due oggetti rappresentano la medesima stringa.

Se vogliamo confrontare gli oggetti come abbiamo fatto nell'applicazione `ConfrontoFrazioni` del paragrafo precedente, e non i riferimenti a essi, dobbiamo utilizzare un metodo appropriato. Per confrontare i suoi oggetti, ogni classe dovrebbe mettere a disposizione un metodo `equals`, come fanno ad esempio la classe `Frazione` e la classe `String`.

Per evidenziare la differenza tra il comportamento del metodo `equals` e quello dell'operatore `==` si consideri la situazione descritta nella Figura 3.1(b). In questo caso l'espressione `u.equals(v)` risulta vera; infatti i due oggetti distinti cui fanno riferimento `u` e `v` modellano la stessa stringa. Al contrario, l'espressione `u == v` risulta falsa, dato che oggetti distinti hanno

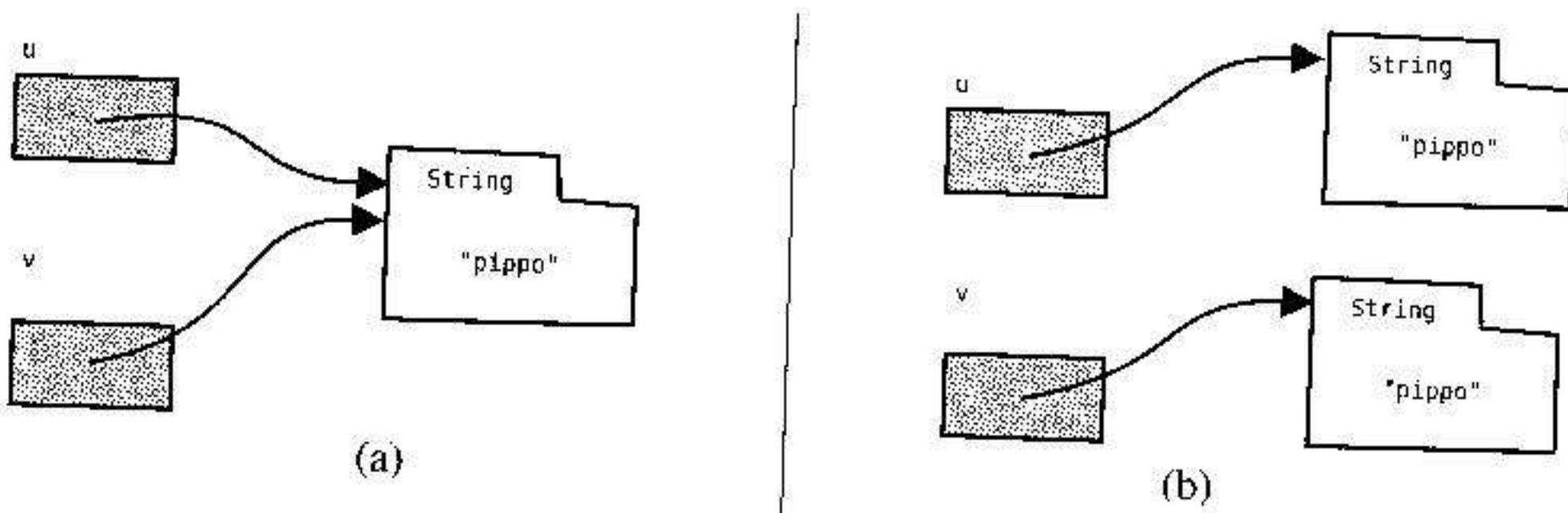


Figura 3.1 Uguaglianza fra riferimenti.

riferimenti diversi. Ovviamente `u.equals(v)` risulta vera anche nella situazione illustrata nella Figura 3.1(a). Per concludere quest'analisi sul metodo `equals` osserviamo che il criterio di uguaglianza fra gli oggetti che tale metodo implementa è strettamente legato a ciò che la classe modella.

3.4 Operatori booleani e condizioni

Un tipo è caratterizzato dai suoi valori e dalle operazioni che si possono compiere su di essi. Vedremo ora quali sono gli operatori che Java mette a disposizione per il tipo `boolean` e come si scrivono le condizioni, cioè le espressioni di tipo `boolean`.

Il tipo `boolean` dispone di due operatori binari indicati con `&&` (*congiunzione o and*) e `||` (*disgiunzione o or*), e di un operatore unario indicato con `!` (*negazione o not*), che, applicati a valori di tipo `boolean`, restituiscono un risultato di tipo `boolean`.

Il risultato della congiunzione è *vero* se e solo se *ambedue* gli operandi hanno valore *vero*; il risultato della disgiunzione è *vero* se e solo se *almeno uno* degli operandi ha valore *vero*. La negazione scambia *vero* con *falso*. In altre parole il comportamento dei tre operatori è definito dalle seguenti *tabelle di verità*.

<code>x</code>	<code>y</code>	<code>x && y</code>	<code>x</code>	<code>y</code>	<code>x y</code>	<code>x</code>		<code>!x</code>
false	false	false	false	false	false	false		true
false	true	false	false	true	true	true		false
true	false	false	true	false	true	true		false
true	true	true	true	true	true	true		false

Utilizzando gli operatori `&&`, `||` e `!` è possibile costruire espressioni booleane complesse.

Nelle espressioni aritmetiche comuni le operazioni vengono eseguite procedendo da sinistra verso destra dando la precedenza, in assenza di parentesi, a moltiplicazione e divisione rispetto a somma e sottrazione. Ad esempio l'espressione $2 + 3 - 4 * 5$ equivale a $(2 + 3) - (4 * 5)$. Le

espressioni booleane sono valutate in modo analogo, tenendo conto del fatto che l'operatore `!` ha la massima precedenza e `||` la minima. Pertanto, se `a`, `b` e `c` sono di tipo `boolean`, l'espressione `!a && b || a && !c` è equivalente a `((!a) && b) || (a && (!c))`.³

Costruiamo ora la tabella di verità dell'espressione `!(a && b) || (a && c)` a partire dalle tabelle dei tre operatori. A sinistra della tabella elenchiamo i valori che possono assumere le variabili `a`, `b` e `c`. A destra, sotto ogni operatore, riportiamo i valori di verità corrispondenti ai valori elencati a sinistra: calcoliamo prima di tutto i valori dell'espressione `a && b` riportandoli sotto il primo `&&`; su di essi applichiamo l'operatore `!` scrivendo nella colonna sottostante i risultati; calcoliamo poi il valore di `a && c` riportando il risultato sotto il secondo `&&`; infine applichiamo l'operatore `||` ai valori scritti nelle colonne sotto il `!` e sotto il secondo `&&` ottenendo così il valore dell'intera espressione, riportato nella colonna sotto l'operatore `||`:

<code>a</code>	<code>b</code>	<code>c</code>	<code>!</code>	<code>(a && b)</code>	<code> </code>	<code>(a && c)</code>
<code>false</code>	<code>false</code>	<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>false</code>
<code>false</code>	<code>false</code>	<code>true</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>true</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>false</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>false</code>
<code>true</code>	<code>false</code>	<code>true</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>	<code>false</code>	<code>false</code>
<code>true</code>	<code>true</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>true</code>	<code>true</code>

Dati due valori booleani `x` e `y`, valgono le seguenti equivalenze, dette *leggi di De Morgan*:

- `!(x && y)` è equivalente a `!x || !y`
- `!(x || y)` è equivalente a `!x && !y`

Come esempio d'uso delle leggi di De Morgan consideriamo una variabile `x` di tipo `int` e supponiamo di voler esprimere, tramite un'espressione Java, la condizione “il valore assoluto di `x` è minore di 4” o, in altre parole, “il valore di `x` è compreso nell'intervallo tra -4 e +4, estremi esclusi”. Quindi, il valore di `x` deve verificare *ambedue* le condizioni: essere maggiore di -4 ed essere minore di +4. La condizione è pertanto:

$$(x > -4) \&\& (x < 4)$$

Consideriamo ora la condizione “il valore assoluto di `x` *non* è minore di 4”. Ciò significa che il valore di `x` non deve trovarsi all'interno dell'intervallo indicato in precedenza, cioè dev'essere minore o uguale a -4 oppure maggiore o uguale a 4. La condizione è dunque:

$$(x \leq -4) \mid\mid (x \geq 4)$$

Si osservi che le due condizioni sono una la negazione dell'altra. È possibile passare dalla negazione della prima condizione alla seconda (e dalla negazione della seconda alla prima) tramite le leggi di De Morgan.

³ Per un riassunto delle precedenze degli operatori si veda l'Appendice A.

Infatti la negazione della prima condizione è:

`!((x > -4) && (x < 4))`

da cui si ottiene

`!(x > -4) || !(x < 4)`

I due `!` possono essere eliminati negando le condizioni considerate (si ricordi che `!(x > y)` equivale a `x <= y`, mentre `!(x >= y)` equivale a `x < y`). In questo modo si ottiene

`(x <= -4) || (x >= 4)`

Si osservi che, quando la prima condizione di una congiunzione `a && b` è falsa, si può concludere immediatamente che l'intera condizione è falsa, senza valutare la seconda condizione. Analogamente, quando la condizione `a` di una disgiunzione `a || b` è vera, si può concludere che la disgiunzione è vera senza valutare la condizione `b`.

Esercizi

3.9 Dimostrate le leggi di De Morgan costruendo le tabelle di verità dei lati a sinistra e a destra delle due uguaglianze.

- (1) $!(x \&\& y) == !x \mid\mid !y$
- (2) $!(x \mid\mid y) == !x \&\& !y$

Scrivete poi un'applicazione Java che stampi a video le tavole di verità dei lati a sinistra e a destra di tali uguaglianze.

3.10 Costruite le tabelle di verità delle seguenti espressioni

- (1) `(a && !b) || !a`
- (2) `!a || b && (a || c)`
- (3) `a && !(c || b)`

dove `a`, `b` e `c` sono di tipo `boolean`.

3.11 Due condizioni sono equivalenti se hanno la stessa tabella di verità. Supponendo che `a` e `b` siano variabili di tipo `int`, riscrivete le seguenti condizioni in una forma equivalente che non contenga l'operatore `!`.

- (1) `!(a == 10) || !(b != 20)`
- (2) `!(a < 10) && !(a > 20)`
- (3) `!((a < 10) && (a > 20))`
- (4) `!((a >= 10) && (a <= 20))`

- (5) $((a > 10) \&\& (a \leq 20))$
- (6) $((a == 5) || !(b < 10))$

3.12 Siano x , y e z variabili di tipo boolean, e a una variabile di tipo int. Tra le seguenti condizioni individuate quelle che sono sempre vere e quelle che sono sempre false. Per ognuna delle rimanenti condizioni, indicate dei valori delle variabili che vi compaiono che rendano la condizione vera e dei valori che rendano la condizione falsa:

- (1) $(x == x) || (x == !x)$
- (2) $(x == x) \&\& (x == !x)$
- (3) $(x == y) || (x == !y)$
- (4) $(x == y) \&\& (x == !y)$
- (5) $z || !z$
- (6) $z \&\& !z$
- (7) $x || (y \&\& !x)$
- (8) $x || (y || !x)$
- (9) $x \&\& (y \&\& !x)$
- (10) $x \&\& (y || !x)$
- (11) $(a \geq 10) \&\& (a \leq 25)$
- (12) $(a \geq 10) || (a \leq 25)$
- (13) $(a < 10) \&\& (a > 25)$
- (14) $(a < 10) || (a > 25)$
- (15) $x || (a != 10)$
- (16) $x \&\& (a != 10)$
- (17) $x || (a != a)$
- (18) $x \&\& (a != a)$
- (19) $x || (a \geq 10) \&\& (a \leq 25)$
- (20) $(x || (a \geq 10)) \&\& (a \leq 25)$
- (21) $x || (a \geq 10) || (a \leq 25)$
- (22) $x \&\& (a \geq 10) || (a \leq 25)$
- (23) $x \&\& ((a \geq 10) || (a \leq 25))$

3.5 I cicli while e do...while

Scriviamo ora un'applicazione che legga una sequenza di frazioni e ne calcoli la somma. Utilizzando i costrutti iterativi, introdotti nel Paragrafo 1.9, possiamo descrivere il procedimento come segue:

```

azzerla somma
ESEGUI
    leggi una frazione
    aggiungi la frazione alla somma
QUANDO ci sono ancora frazioni da leggere
    scrivi la somma

```

La somma di frazioni è una frazione: pertanto essa sarà memorizzata in un oggetto della classe **Frazione**. A tale scopo dichiariamo una variabile riferimento di tipo **Frazione**, che inizialmente si riferirà a un oggetto che rappresenta la frazione di valore zero. Pertanto definiamo una variabile **somma** come segue:

```
Frazione somma = new Frazione(0);
```

Codifichiamo ora le istruzioni da ripetere all'interno del ciclo supponendo di avere dichiarato una variabile **in** di tipo **ConsoleInputManager**, due variabili **num** e **den** di tipo **int** e una variabile **nuovaFrazione** di tipo **Frazione**:

```

num = in.readInt("Numeratore? ");
den = in.readInt("Denominatore? ");
nuovaFrazione = new Frazione(num, den);
somma = somma.piu(nuovaFrazione);

```

Si noti che, per risolvere il problema proposto, da questo punto in poi del programma il riferimento alla frazione letta non è più necessario. Pertanto si può scrivere il codice in maniera più compatta utilizzando l'espressione di creazione direttamente come argomento del metodo **piu**:

```

num = in.readInt("Numeratore? ");
den = in.readInt("Denominatore? ");
somma = somma.piu(new Frazione(num, den));

```

Come osservato in precedenza, questa sequenza di istruzioni dev'essere *ripetuta* per ogni nuova frazione aggiunta alla sequenza.

In Java l'iterazione con il controllo in coda è realizzata mediante l'istruzione **do...while**, che ha questa forma:

```

do
    istruzione
    while (condizione)

```

Dove:

- *condizione* è un'espressione di tipo boolean;
- *istruzione* è l'istruzione che dev'essere ripetuta: essa può essere un'istruzione singola oppure un blocco di istruzioni.

L'esecuzione avviene come segue.

- (1) Viene eseguito il corpo del ciclo, cioè *istruzione*.
- (2) Viene valutata l'espressione *condizione*.
- (3) Se la condizione è vera (il suo valore è `true`), l'esecuzione prosegue dal Punto (1).
- (4) Se la condizione è falsa (il suo valore è `false`), il ciclo termina e l'esecuzione riprende dalla prima istruzione che segue l'istruzione `do...while`.

Si osservi che l'esecuzione del ciclo termina quando la condizione risulta falsa; poiché il controllo avviene in coda, il corpo del ciclo è sempre eseguito almeno una volta.

```
do {  
    num = in.readInt("Numeratore? ");  
    den = in.readInt("Denominatore? ");  
    somma = somma.piu(new Frazione(num, den));  
} while (...);
```

A questo punto dobbiamo individuare una condizione opportuna per decidere quando interrompere il ciclo. Scegliamo di chiedere all'utente, a ogni iterazione, se desidera procedere all'inserimento di un'ulteriore frazione. A questo scopo possiamo utilizzare il metodo `readSiNo` della classe `ConsoleInputManager`. Questo metodo chiede all'utente di effettuare una scelta inserendo uno dei due caratteri: `s` o `n`. Il metodo restituisce un boolean `e`, per la precisione, restituisce `true` se il carattere inserito è `s`, restituisce `false` se il carattere inserito è `n`. Possiamo memorizzare il risultato restituito dal metodo in una variabile `continua` di tipo boolean e utilizzarla per decidere se continuare l'esecuzione del ciclo o no:

```
do {  
    num = in.readInt("Numeratore? ");  
    den = in.readInt("Denominatore? ");  
    somma = somma.piu(new Frazione(num, den));  
    continua = in.readSiNo("Vuoi inserire un'altra frazione (s/n)? ");  
} while (continua);
```

Ecco il codice di una classe `SommaSequenza`, sviluppata secondo quanto abbiamo appena detto:

```
import prog.io.*;  
import prog.utili.Frazione;
```

```

class SommaSequenza {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int num, den;
        Frazione somma = new Frazione(0);
        boolean continua;

        do {
            num = in.readInt("Numeratore? ");
            den = in.readInt("Denominatore? ");
            somma = somma.piu(new Frazione(num, den));
            continua = in.readSiNo("Vuoi inserire un'altra frazione (s/n)? ");
        } while (continua);

        out.println("La somma è " + somma.toString());
    }
}

```

Consideriamo ora lo schema di iterazione con controllo in testa QUANDO...RIPETI, presentato nel Paragrafo 1.9. In Java questo schema viene implementato mediante l'istruzione `while`, la cui forma è:

```

while (condizione)
    istruzione

```

Dove:

- *condizione* è un'espressione booleana scritta obbligatoriamente tra parentesi tonde
- *istruzione* è l'istruzione che dev'essere ripetuta; essa può essere un'istruzione singola oppure un blocco di istruzioni.

L'esecuzione di un'istruzione `while` procede come segue.

- (1) Viene valutata l'espressione *condizione*.
- (2) Se la condizione è vera (il suo valore è `true`):
 - viene eseguita l'istruzione nel corpo del ciclo;
 - l'esecuzione continua dal Punto (1).

- (3) Se la condizione è falsa (il suo valore è `false`), il ciclo termina e l'esecuzione continua dalla prima istruzione che segue il ciclo `while`.

Si ricordi che l'esecuzione del ciclo termina quando *condizione* risulta falsa. Inoltre, poiché il controllo della condizione avviene prima dell'esecuzione del corpo del ciclo, *istruzione* può essere eseguita anche zero volte.

Presentiamo ora una nuova versione dell'applicazione SommaSequenza, in cui sostituiamo il ciclo `do...while` con un ciclo `while`. Con il semplice spostamento del controllo in testa otterremo il ciclo:

```
while (continua) {
    num = in.readInt("Numeratore? ");
    den = in.readInt("Denominatore? ");
    somma = somma.piu(new Frazione(num, den));
    continua = in.readSiNo("Vuoi inserire un'altra frazione (s/n)? ");
}
```

La variabile `continua`, dichiarata immediatamente prima del ciclo, al momento della prima valutazione della condizione non è inizializzata; pertanto il compilatore segnala un errore sulla condizione del ciclo `while`. Per ottenere lo stesso risultato della versione precedente, in cui viene sempre letta almeno una frazione, è sufficiente inizializzare a `true` la variabile `continua` prima del ciclo:

```
boolean continua = true;

while (continua) {
    num = in.readInt("Numeratore? ");
    den = in.readInt("Denominatore? ");
    somma = somma.piu(new Frazione(num, den));
    continua = in.readSiNo("Vuoi inserire un'altra frazione (s/n)? ");
}
```

Anziché inizializzare direttamente la variabile `continua` a `true`, per leggere almeno una frazione possiamo chiedere all'utente se intende effettuare il primo inserimento oppure no: a tale scopo richiamiamo il metodo `readSiNo`:

```
boolean continua = in.readSiNo("Vuoi inserire una frazione (s/n)? ";

while (continua) {
    num = in.readInt("Numeratore? ");
    den = in.readInt("Denominatore? ");
    somma = somma.piu(new Frazione(num, den));
    continua = in.readSiNo("Vuoi inserire un'altra frazione (s/n)? ");
}
```

In questo caso, se la risposta dell'utente è negativa, alla variabile `continua` è assegnato `false`; pertanto il corpo del ciclo non viene eseguito e l'esecuzione riprende dall'istruzione che lo segue, cioè dalla comunicazione del risultato (che in questo caso sarà il valore zero).

Modifichiamo ora l'applicazione con questa nuova versione del ciclo, in modo che, oltre alla somma delle frazioni, sia comunicata anche la loro media. Per il calcolo della media è necessario conoscere, alla fine del ciclo, il numero di frazioni lette. A tale scopo utilizziamo come contatore una variabile `contaFrazioni` di tipo `int`, inizializzata a zero e incrementata a ogni iterazione. La media sarà la frazione ottenuta dividendo `somma` per la frazione $\frac{1}{\text{contaFrazioni}}$. Il codice completo dell'applicazione è il seguente:

```
import prog.io.*;
import prog.utili.Frazione;

class MediaSequenza {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int num, den;
        Frazione somma = new Frazione(0);
        int contaFrazioni = 0;
        boolean continua = in.readSiNo("Vuoi inserire una frazione (s/n)? ");

        while (continua) {
            num = in.readInt("Numeratore? ");
            den = in.readInt("Denominatore? ");
            somma = somma.piu(new Frazione(num, den));
            contaFrazioni = contaFrazioni + 1;
            continua = in.readSiNo("Vuoi inserire un'altra frazione (s/n)? ");
        }

        if (contaFrazioni == 0)
            out.println("Non è stata inserita alcuna frazione");
        else {
            Frazione media = somma.diviso(new Frazione(contaFrazioni));
            out.println("La somma è " + somma.toString()+
                "; la media è " + media.toString());
        }
    }
}
```

Una delle forme più semplici di uso del ciclo `while` è quella in cui la condizione del ciclo è costituita da un letterale di tipo `boolean`. Un ciclo in cui la condizione sia costituita dal solo letterale `false` non risulta utile, perché le istruzioni che si trovano all'interno del suo corpo non saranno mai eseguite. Al contrario, un ciclo `while` in cui la condizione è costituita dal solo letterale `true` è un ciclo infinito, cioè un ciclo che viene potenzialmente eseguito un numero infinito di volte.⁴ Potremmo utilizzare tale ciclo per realizzare una variante dell'applicazione Pappagallo, che si comporti come un instancabile pappagallo, nel modo seguente:

```
import prog.io.*;

class PappagalloInstancabile {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager tastiera = new ConsoleInputManager();
        ConsoleOutputManager video = new ConsoleOutputManager();

        String messaggio;
        String risposta;

        //ciclo infinito
        while (true) {
            messaggio = tastiera.readLine();
            risposta = messaggio.toUpperCase();
            video.println(risposta);
        }
    }
}
```

In questo caso, nel corpo del ciclo vengono eseguite due operazioni: prima viene letta da tastiera una stringa, quindi la stringa viene stampata a video con tutti i caratteri maiuscoli. Dato però che non è possibile modificare il valore della condizione fra due esecuzioni successive del corpo del ciclo (`true` è un'espressione costante), il programma continuerà a eseguire tali operazioni all'infinito. In questa situazione si dice anche che il programma è entrato in un *loop infinito*. L'unico modo di interrompere l'esecuzione è quello di utilizzare la combinazione di tasti `CTRL + C`, che non interagisce con l'applicazione ma interrompe l'esecuzione della Java Virtual Machine. Questo non è un modo elegante di procedere: un programma che si trovi a eseguire un loop infinito è un programma scritto in modo scorretto; un ciclo dovrebbe sempre terminare. Nel caso della nostra applicazione sarebbe opportuno prevedere una stringa il cui inserimento determini la “volontaria” conclusione dell'applicazione.

⁴ Dovrebbe essere chiaro che esistono cicli infiniti la cui condizione non è un semplice letterale. Ne forniremo svariati esempi negli esercizi di questo capitolo.

Ad esempio, possiamo prevedere che l'applicazione termini modificando il ciclo in questo modo:

```
while (!messaggio.equals("stanco")) {
    messaggio = tastiera.readLine();
    risposta = messaggio.toUpperCase();
    video.println(risposta);
}
```

In questo caso, le istruzioni nel corpo del ciclo vengono ripetute fintantoché la stringa letta è diversa da "stanco". Infatti, la condizione:

```
!messaggio.equals("stanco")
```

è falsa quando la stringa cui fa riferimento `messaggio` è uguale a "stanco". Si osservi che la modifica che abbiamo apportato al ciclo utilizza la variabile `messaggio` nella condizione del ciclo. Al momento della prima valutazione di tale condizione, alla variabile `messaggio` non risulta assegnato alcun valore, cioè `messaggio` non risulta inizializzata. Il compilatore Java si preoccupa di garantire che, quando viene utilizzato il valore di una variabile, vi sia un'operazione precedente che le abbia assegnato un valore. Nel nostro esempio non è così, in quanto la variabile `messaggio` con l'istruzione

```
String messaggio;
```

risulta dichiarata ma non definita (non è stata inizializzata); infatti non esiste alcun assegnamento a essa prima del suo utilizzo nella condizione. Conseguentemente il compilatore segnala questo errore:

```
PappagalloStanco.java:13: variable messaggio might not have been
initialized
    while (!messaggio.equals("stanco")) {
    ^
1 error
```

Si noti che il compilatore segnala la riga in cui ha individuato l'errore dopo il nome del file dove si trova l'errore (in questo caso alla riga 13 del file `PappagalloStanco.java`) e indica anche, con il simbolo ^, il punto della riga in cui ha individuato l'errore. Possiamo eliminare l'errore fornendo un'inizializzazione della variabile. In questo caso possiamo inizializzarla assegnandole la stringa vuota nel modo seguente:

```
String messaggio = "";
```

In questo modo, al momento della prima valutazione della condizione, il valore di `messaggio` è definito ed è senz'altro diverso dalla stringa "stanco". Per comodità riportiamo il codice completo dell'applicazione:

```
import prog.io.*;

class PappagalloStanco {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager tastiera = new ConsoleInputManager();
        ConsoleOutputManager video = new ConsoleOutputManager();

        String messaggio = "";
        String risposta;

        while (!messaggio.equals("stanco")) {
            messaggio = tastiera.readLine();
            risposta = messaggio.toUpperCase();
            video.println(risposta);
        }
    }
}
```

Se in luogo della dichiarazione

```
String messaggio = "";
```

avessimo usato la dichiarazione

```
String messaggio = null;
```

in fase di esecuzione la Java Virtual Machine avrebbe segnalato il seguente errore in corrispondenza della condizione del ciclo:

```
Exception in thread "main" java.lang.NullPointerException
at PappagalloStanco.main(PappagalloStanco.java:13)
```

L'errore è dovuto al tentativo di inviare un messaggio a un oggetto inesistente: si tenta infatti di invocare il metodo (`equals`) dell'oggetto riferito dalla variabile (`messaggio`) che, però, contiene `null`.

3.6 L'istruzione for

In questo paragrafo presentiamo l'istruzione `for`, che fornisce un'altra struttura di controllo per l'iterazione. Il ciclo `for` è utilizzato principalmente quando l'iterazione è basata su una variabile cui viene assegnato un valore all'inizio del ciclo, che viene modificata a ogni iterazione, e che viene utilizzata all'interno della condizione del ciclo.

Supponiamo ad esempio di voler visualizzare la sequenza di numeri da 1 a 10. Ipotizzando che `out` contenga un riferimento a un oggetto di tipo `ConsoleOutputManager` e `cont` sia una variabile di tipo `int`, possiamo visualizzare la sequenza tramite la seguente istruzione:

```
for (cont = 1; cont <= 10; cont = cont + 1)
    out.println(cont);
```

L'intestazione del ciclo `for`, costituita dalla parte tra parentesi tonde che segue la parola riservata `for`, contiene, separati da punti e virgola, i tre elementi che seguono:

- (1) un'istruzione che viene eseguita una sola volta, prima di tutte le altre, e che provvede in genere all'inizializzazione delle variabili coinvolte nel ciclo, in questo caso l'inizializzazione della variabile contatore `cont = 1;`
- (2) una condizione che viene valutata *all'inizio di ogni iterazione*; l'esecuzione del ciclo termina quando la condizione risulta falsa;
- (3) un'istruzione che sarà eseguita *al termine di ciascuna iterazione*, prima di valutare di nuovo la condizione.

Al fine di rendere il codice leggibile, è *opportuno* che le istruzioni da ripetere all'interno del ciclo non modifichino la variabile *di controllo* del ciclo, cioè in questo caso la variabile `cont`.

Possiamo schematizzare il ciclo `for` come segue:

```
for (espr_inizializzazione; condizione; espr_incremento)
    istruzione
```

Dove:

- *espr_inizializzazione* è una lista di espressioni, separate da virgola. Sono le espressioni di inizializzazione delle variabili di controllo del ciclo e possono eventualmente contenere anche la dichiarazione delle variabili,
- *condizione* è una qualunque espressione booleana,
- *espr_incremento* è una lista di espressioni,
- *istruzione* è una singola istruzione oppure un blocco di istruzioni.

Tutte le componenti di un ciclo `for`, cioè *espr_inizializzazione*, *condizione*, *espr_incremento* e *istruzione* sono opzionali. L'esecuzione del ciclo avviene come segue.

- (1) Per prima cosa vengono valutate le espressioni che compaiono in *espr_inizializzazione*.
- (2) Quindi viene valutata l'espressione *condizione*.
- (3) Se la condizione è vera (il suo valore è `true`):
 - viene eseguito il blocco di istruzioni nel corpo del ciclo;

- vengono valutate le espressioni che compaiono in *expr_incremento*;
 - l'esecuzione prosegue dal Punto (2).
- (4) Se la condizione è falsa (il suo valore è `false`), l'esecuzione riprende dalla prima istruzione che segue l'istruzione `for`.

L'espressione di inizializzazione può contenere direttamente la dichiarazione della variabile di controllo del ciclo, come in questo esempio:

```
for (int cont = 1; cont <= 10; cont = cont + 1)
    out.println(cont);
```

In questo caso la variabile `cont` *non esiste* al di fuori del ciclo.

Il ciclo `for` potrebbe essere utilizzato in altre forme, ad esempio tralasciando una delle parti introdotte sopra. Nel ciclo che segue mancano sia le espressioni di inizializzazione sia le espressioni di incremento:

```
for ( ; x != 0; ) {
    x = in.readInt("Inserisci un numero ");
    out.println(x);
}
```

Si tratta comunque di un impiego improprio del ciclo `for`; in casi come questo è preferibile utilizzare il ciclo `while`. Per rendere il codice leggibile, il ciclo `for` dovrebbe essere utilizzato solo quando è possibile evidenziare una variabile di controllo; inoltre le istruzioni da ripetere all'interno del ciclo non dovrebbero modificare tale variabile. Il modo in cui avviene l'iterazione risulta evidente leggendo l'intestazione del ciclo.

Come primo esempio d'uso del ciclo `for` riscriviamo ora l'applicazione `Cornice`, presentata nel Paragrafo 2.3, in modo tale che generi le stringhe `rigaCornice` e `spazi` utilizzando cicli `for` anziché il metodo `substring`. A tal fine è sufficiente osservare che la stringa `rigaCornice` dev'essere costituita da `lunghezza + 6` caratteri `*`, dove `lunghezza` è la lunghezza della stringa letta in input. La stringa `spazi` dev'essere invece costituita da `lunghezza + 4` spazi. Possiamo quindi generarle utilizzando il ciclo `for` e l'operatore di concatenazione fra stringhe nel modo seguente:

```
String rigaCornice = "";
for (int i = 0; i < lunghezza + 6; i = i + 1)
    rigaCornice = rigaCornice.concat("*");

String spazi = "";
for (int i = 0; i < lunghezza + 4; i = i + 1)
    spazi = spazi.concat(" "));
```

Il testo completo dell'applicazione è ora questo:

```
import prog.io.*;

class Cornice {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        String frase;
        int lunghezza;

        //lettura del nome e calcolo della lunghezza
        frase = in.readLine("Inserisci il tuo nome> ");
        lunghezza = frase.length();

        //costruzione delle stringhe che costituiranno la cornice
        String rigaCornice = "";
        for (int i = 0; i < lunghezza + 6; i = i + 1)
            rigaCornice = rigaCornice.concat("*");
        String spazi = "";
        for (int i = 0; i < lunghezza + 4; i = i + 1)
            spazi = spazi.concat(" ");
        String rigaIntermedia, rigaTesto;
        rigaIntermedia = "*" + spazi + "*";
        rigaTesto = "* " + frase + " *";

        //visualizzazione della cornice
        out.println(rigaCornice);
        out.println(rigaIntermedia);
        out.println(rigaTesto);
        out.println(rigaIntermedia);
        out.println(rigaCornice);
    }
}
```

Osserviamo che l'inizializzazione delle variabili `rigaCornice` e `spazi` al momento della loro dichiarazione è essenziale.

Se infatti non le avessimo inizializzate, a quale stringa avremmo concatenato il primo * e il primo spazio nei due cicli? Per evitare possibili problemi legati all'uso di variabili non inizializzate, il compilatore Java controlla che quando in un'espressione viene utilizzata una variabile questa contenga un valore significativo. Ciò è garantito solo se tale impiego è preceduto da un'istruzione di assegnamento di valore alla variabile. Nel nostro caso il valore viene assegnato alle

variabili `rigaCornice` e `spazi` direttamente in fase di dichiarazione. Se, ad esempio, ci fossimo limitati a dichiarare la variabile `rigaCornice` anziché definirla, cioè se avessimo utilizzato l'istruzione:

```
String rigaCornice;
```

il compilatore avrebbe segnalato i seguenti errori:

```
Cornice.java:20: variable rigaCornice might not have been initialized
    rigaCornice = rigaCornice.concat("*");
                           ^
```

```
Cornice.java:29: variable rigaCornice might not have been initialized
    out.println(rigaCornice);
                           ^
```

2 errors

Esempio: stringhe palindrome

Vogliamo costruire una classe contenente un metodo `main` in grado di riconoscere le stringhe *palindrome*, cioè le stringhe che, lette al contrario, rimangono identiche. Esempi di stringhe palindrome sono `radar`, `anna` e `ailatiditalia`.

Il metodo può essere strutturato come segue:

```
lettura della stringa
esame della stringa
scrittura del risultato
```

Supponiamo che, dopo la fase di lettura, in una variabile `s` di tipo `String` vi sia un riferimento alla stringa da analizzare.

Sviluppiamo la fase di esame della stringa in modo che il risultato sia posto in una variabile di tipo `boolean`, denominata `palindroma`. In altre parole vogliamo che dopo la fase di esame della stringa, la variabile `palindroma` contenga valore `true` se e solo se la stringa riferita da `s` è palindroma.

Per analizzare la stringa possiamo scandirla da sinistra verso destra confrontando il primo carattere con l'ultimo, il secondo con il penultimo, e così via. Se i caratteri confrontati coincidono ogni volta, la stringa è palindroma; altrimenti, se in un confronto troviamo due caratteri differenti, la stringa non lo è.

In Java i caratteri sono rappresentati tramite il tipo primitivo `char`. La classe `String` fornisce un metodo di nome `charAt`, utile per estrarre caratteri da una stringa. Il metodo riceve come parametro un valore di tipo `int`, che rappresenta una posizione all'interno della stringa, e restituisce un valore di tipo `char` uguale al carattere presente in tale posizione.

Possiamo inizializzare la variabile `palindroma` a `true`. Se i due caratteri confrontati sono diversi, le assegniamo `false`. Ricordando che le posizioni all'interno delle stringhe vengono contate a partire da zero, possiamo scrivere questo schema:

```

boolean palindroma = true;
int lung = s.length();
for (int i = 0; i < lung; i = i + 1)
    if (i caratteri di s nelle posizioni i e lung - i - 1 sono diversi)
        palindroma = false;

```

Per ottenere un singolo carattere della stringa ricorriamo al metodo `charAt`. Ecco il codice completo di una classe `Palindromi` contenente un metodo `main` costruito secondo questo schema:

```

import prog.io.*;

//determina se una stringa è palindroma
class Palindromi {

    public static void main(String [] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //lettura della stringa
        String s = in.readLine("Inserire la stringa da esaminare ");

        //esame della stringa
        boolean palindroma = true;
        int lung = s.length();
        for (int i = 0; i < lung; i = i + 1)
            if (s.charAt(i) != s.charAt(lung - i - 1))
                palindroma = false;

        //comunicazione del risultato
        if (palindroma)
            out.println("La stringa " + s + " è palindroma");
        else
            out.println("La stringa " + s + " non è palindroma");
    }
}

```

Possiamo osservare che il ciclo `for` precedente fa una serie di confronti inutili; infatti, quando l'indice `i` raggiunge la metà della stringa, i confronti che vengono fatti sono ripetizioni di confronti già effettuati in precedenza. Inoltre l'uso dell'espressione `lung - i - 1` rende il codice poco leggibile.

Un modo più intuitivo per scrivere il ciclo consiste nell'immaginare di avere due indici, uno che parte dall'inizio della stringa, l'altro dalla fine. I due indici sono spostati, di volta

in volta, verso il centro della stringa; quando si incontrano, il procedimento termina. A ogni passo si confrontano i caratteri nelle posizioni corrispondenti ai due indici. In questo caso, nel ciclo occorrono due istruzioni di inizializzazione e due istruzioni di aggiornamento. È possibile inserirle nell'intestazione del ciclo per mezzo della virgola.

Ecco la nuova versione del ciclo:

```
boolean palindroma = true;
for (int sx = 0, dx = s.length() - 1; sx < dx; sx = sx + 1,
    dx = dx - 1)
    if (s.charAt(sx) != s.charAt(dx))
        palindroma = false;
```

Ed ecco la nuova versione della classe Palindromi:

```
import prog.io.*;

//determina se una stringa è palindroma
class Palindromi {

    public static void main(String [] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //lettura della stringa
        String s = in.readLine("Inserire la stringa da esaminare ");

        //esame della stringa
        boolean palindroma = true;
        for (int sx = 0, dx = s.length() - 1; sx < dx; sx = sx + 1,
            dx = dx - 1)
            if (s.charAt(sx) != s.charAt(dx))
                palindroma = false;

        //comunicazione del risultato
        if (palindroma)
            out.println("La stringa " + s + " è palindroma");
        else
            out.println("La stringa " + s + " non è palindroma");
    }
}
```

3.7 Le istruzioni break e continue

L'istruzione `break` può essere utilizzata per provocare l'uscita dai cicli. Consideriamo ad esempio la seconda versione dell'applicazione `Palindromi`; il ciclo `for` impiegato è basato su due indici, `sx` e `dx`, che indicano le posizioni corrispondenti della stringa da confrontare partendo dalla prima e dall'ultima, e muovendosi verso il centro. Quando si individua una coppia di posizioni corrispondenti che contengono caratteri differenti, si può immediatamente concludere che la stringa non è palindroma. Nel codice ciò avviene assegnando `false` alla variabile `palindroma`. A questo punto si potrebbe terminare l'esecuzione del ciclo evitando i confronti successivi. Possiamo farlo introducendo un'istruzione `break`. In particolare il ciclo può essere riscritto come:

```
for (int sx = 0, dx = s.length() - 1; sx < dx; sx = sx + 1,
    dx = dx - 1)
    if (s.charAt(sx) != s.charAt(dx)) {
        palindroma = false;
        break;
    }
```

Un altro esempio d'uso dell'istruzione `break` è dato dalla seguente versione dell'applicazione `PappagalloStanco`:

```
import prog.io.*;

class PappagalloStanco {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager tastiera = new ConsoleInputManager();
        ConsoleOutputManager video = new ConsoleOutputManager();

        String messaggio;
        String risposta;

        do {
            messaggio = tastiera.readLine();
            risposta = messaggio.toUpperCase();
            video.println(risposta);
            if (messaggio.equals("stanco"))
                break;
        } while (true);
    }
}
```

Questa versione dell'applicazione, come quella presentata prima, continua a ripetere le stringhe inserite in `input` dall'utente finché non è inserita la stringa "stanco". In questo caso l'esecuzione del ciclo non viene però interrotta quando la condizione del ciclo diventa falsa (infatti la condizione è costituita dalla costante `true`), ma quando è eseguito il corpo dell'`if`, che contiene appunto l'istruzione `break`. Si osservi che, benché le due applicazioni si comportino allo stesso modo, quest'ultima versione è scritta in maniera più contorta. Anziché essere evidenziate nella sua condizione, le ragioni per le quali il ciclo termina sono nascoste nel corpo del ciclo.

Un'altra istruzione utilizzata nei cicli è l'istruzione `continue`. Essa provoca l'interruzione dell'esecuzione del blocco di istruzioni interne al ciclo e il passaggio all'iterazione successiva. In particolare, nel caso dei cicli `while` o `do...while`, incontrando un'istruzione `continue` si saltano le restanti istruzioni nel corpo del ciclo e si passa immediatamente alla valutazione della condizione e, in base a questa, all'iterazione successiva o all'uscita dal ciclo. Nel caso dei cicli `for`, invece, quando si incontra un'istruzione `continue` si passa a eseguire l'istruzione di incremento del ciclo e poi alla valutazione della condizione.

Il ciclo che segue calcola la somma dei numeri pari presenti in una sequenza di interi, letta da `input`, che termina con zero:

```
int x, somma = 0;
do {
    x = in.readInt();
    if (x == 0)
        break;
    if (x % 2 == 1)
        continue;
    somma = somma + x;
} while (true);
```

Tuttavia il codice scritto in questa forma è poco leggibile. La condizione del ciclo non è evidenziata dopo la parola `while`, ma dev'essere individuata leggendo le istruzioni all'interno del codice. Poiché le istruzioni `break` e `continue` possono ridurre la leggibilità dei programmi, è opportuno farne un uso estremamente limitato, riservato alle situazioni in cui esse permettono effettivamente di semplificare molto la struttura del codice o di aumentarne l'efficienza. Ad esempio, per riconoscere le parole palindrome nel ciclo `for` precedente l'istruzione `break` si rivela vantaggiosa, in quanto consente di eliminare iterazioni inutili. Al contrario, il ciclo riportato sopra per il calcolo della somma dei numeri pari in una sequenza è poco leggibile, e potrebbe essere riscritto facilmente in una forma migliore, che eviti l'uso di `break` e `continue`.

Esercizi

- 3.13 Scrivete un'applicazione che, ricevendo in ingresso un numero intero positivo, indichi se tale numero è primo. Costruite diverse versioni. In una prima versione provate a dividere il numero per tutti i numeri minori mediante un ciclo `for`, al fine di verificare l'esistenza di un divisore. In un'altra versione interrompete l'iterazione quando è stato individuato un

divisore. In un'ulteriore versione riducete il numero massimo di iterazioni osservando che se un numero ammette un divisore, allora ammette anche un divisore minore o uguale alla propria radice quadrata.

- 3.14 Scrivete un'applicazione che legga due interi a e b e stampi tutti i numeri compresi tra a e b per cui la lunghezza della stringa di caratteri che esprime il numero nella lingua italiana è pari alla metà del numero stesso (si veda l'Esercizio 3.8). Un esempio di esecuzione potrebbe essere il seguente.

```
Estremo inferiore? 2
Estremo superiore? 9
6
8
```

- 3.15 Scrivete un'applicazione che, letta una sequenza di frazioni, determini la frazione minore e la frazione maggiore che compaiono nella sequenza, e le stampi a video.
- 3.16 Scrivete un'applicazione che legga una frazione e un esponente (numero intero) e calcoli la frazione ottenuta elevando la frazione data all'esponente. Ad esempio, se viene letta la frazione $2/3$ e l'esponente è 3 , il risultato dovrà essere la frazione $8/27$, se invece viene letta la frazione $2/3$ e l'esponente è -3 , il risultato dovrà essere la frazione $27/8$.
- 3.17 Utilizzando il metodo `compareTo` della classe `String` descritto nell'Esercizio 3.7 scrivete un'applicazione che, letta in ingresso una sequenza di stringhe, individui la stringa minore e la stringa maggiore che compaiono nella sequenza rispetto all'ordinamento lessicografico, e le stampi a video.
- 3.18 Considerate i seguenti metodi `main`. Per ognuno di essi indicate se ci saranno errori in fase di compilazione, se ci saranno errori in fase d'esecuzione (in tal caso indicate dei valori di ingresso che causino errore) e se il metodo può entrare in un ciclo infinito (in tal caso indicate dei valori di ingresso per i quali ciò avviene). Giustificate le risposte.

```
(1) public static void main(String[] args) {
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();
    int x = in.readInt();
    for (int i = 10; i >= x; i = i + 1)
        out.println(i);
}

(2) public static void main(String[] args) {
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();
    int x = in.readInt();
    for (int i = 10; i != x; i = i + 1)
```

```
        out.println(i);
    }

(3) public static void main(String[] args) {
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();
    int x = in.readInt();
    for (int i = 1; i <= x; i = i + 2)
        out.println(i);
}

(4) public static void main(String[] args) {
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();
    int x = in.readInt();
    int y = in.readInt();
    while (x != y) {
        x = x + 1;
        y = y - 1;
    }
    out.println(x);
}

(5) public static void main(String[] args) {
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();
    int x = in.readInt();
    int y = in.readInt();
    if (x < y) {
        while (x != y) {
            x = x + 1;
            y = y - 1
        }
    }
    out.println(x);
}

(6) public static void main(String[] args) {
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();
    int x, y;
    x = in.readInt();
    y = 1;
    do {
        x = x - 2;
        y = y + 1;
```

```

        out.println(x + "" + y);
    }
    while (x != y);
}

```

3.19 Individuate il comportamento dei seguenti metodi:

- (1)

```

public static void main(String[] args) {
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();
    int numero = 0;
    while (numero < 4)
        numero = numero + 1;
    out.println(numero);
}
```
- (2)

```

public static void main(String[] args) {
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();
    int numero = 0;
    while (numero < 4)
        out.println(numero);
    numero = numero + 1;
}
```
- (3)

```

public static void main(String[] args) {
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();
    int numero = 0;
    while (numero < 4);
        numero = numero + 1;
    out.println(numero);
}
```

(4) Scrivete un metodo main che utilizzando un ciclo while visualizzi i numeri da 1 a 4.

3.20 Riscrivete i seguenti frammenti di codice sostituendo i cicli for con cicli while (tutte le variabili utilizzate sono di tipo int):

- (1)

```

for (i = 1; i <= x; i = i + 2)
    out.println(i);
```
- (2)

```

for (i = 10; i != -10; i = i - 2)
    out.println(i);
```
- (3)

```

for (i = 100; i > 0; i = i / 2)
    out.println(i);
```

- 3.21 Scrivete un programma che, letto un intero $n \geq 1$, visualizzi la frazione il cui valore è dato dalla somma $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$.
- 3.22 Scrivete un programma che, letto un intero $n \geq 1$, visualizzi la frazione il cui valore è dato dalla somma $1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{n^2}$.
- 3.23 Scrivete un programma che, letto un intero n qualunque, visualizzi la frazione il cui valore è dato dalla somma $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$ nel caso n sia positivo, la frazione il cui valore è dato dalla somma $-1 + \frac{1}{-2} + \frac{1}{-3} + \dots + \frac{1}{n}$ nel caso n sia negativo, zero nel caso n sia zero.
- 3.24 Siano x e y due variabili di tipo int. Per ognuno dei seguenti frammenti di codice indicate se il ciclo che vi compare è finito oppure infinito. Nel caso il ciclo termini, indicate i valori contenuti nelle variabili x e y al termine dell'esecuzione del frammento di codice.

```
(1)  x = 9;
    y = 9;
    while (x != 0) {
        x = x + 1;
        y = y - 1;
    }

(2)  x = 9;
    y = 9;
    while (x != 0) {
        x = x - 1;
        y = y + 1;
    }

(3)  x = 9;
    y = 9;
    while (x != 0) {
        x = x - 2;
        y = y + 1;
    }

(4)  x = 9;
    y = 9;
    while (x == 0) {
        x = x + 1;
        y = y - 1;
    }

(5)  x = 9;
    y = 9;
    do {
        x = x + 1;
```

```
        y = y - 1;
    } while (x == 0);
(6)  x = 1;
    y = 1;
    while (x == 0) {
        x = x - 1;
        y = y + 1;
    }
(7)  x = 1;
    y = 1;
    do {
        x = x - 1;
        y = y + 1;
    } while (x == 0);
(8)  x = 0;
    y = 0;
    while (x == 0) {
        x = x - 1;
        y = y + 1;
    }
(9)  x = 0;
    y = 0;
    do {
        x = x - 1;
        y = y + 1;
    } while (x == 0);
(10) x = 5;
    y = 0;
    do {
        x = x - 1;
        y = y + 1;
    } while (x == y);
(11) x = 5;
    y = 0;
    do {
        x = x - 1;
        y = y + 1;
    } while (x < y);
(12) x = 5;
    y = 0;
    do {
```

```
x = x - 1;  
y = y + 1;  
} while (x != y);  
  
(13) x = 4;  
y = 0;  
do {  
    x = x - 1;  
    y = y + 1;  
} while (x != y);
```

- 3.25 Siano x e y due variabili di tipo `int`. Per ognuno dei seguenti frammenti di codice stabilite se il ciclo che vi compare può essere infinito e, in caso affermativo, indicate dei valori delle variabili x e y , prima dell'esecuzione del ciclo, per i quali ciò accade; stabilite anche se il ciclo può terminare e, in caso affermativo, indicate dei valori delle variabili x e y , prima dell'esecuzione del ciclo, per i quali ciò accade.

```
(1) do {  
    x = x + 2;  
    y = y - 3;  
} while (x < y);  
  
(2) do {  
    x = x + 2;  
    y = y - 3;  
} while (x != y);  
  
(3) while (x != 0)  
    x = x / 2;  
  
(4) while (x != 0) {  
    x = (x + y) / 2;  
    y = x;  
}  
  
(5) while (x != 0) {  
    if (x < y)  
        x = y;  
    else  
        x = x / y;  
}  
  
(6) while (x != 0) {  
    if (x < y)  
        x = y;  
    else  
        x = y / x;  
}
```

3.26 Considerate l'Esercizio 2.9. Oltre ai metodi ivi descritti la classe Importo mette a disposizione anche i seguenti metodi:

- `public boolean isMaggiore(Importo x)`

Restituisce `true` se l'importo rappresentato dall'oggetto che esegue il metodo è maggiore di quello specificato tramite l'argomento, `false` in caso contrario.

- `public boolean isMinore(Importo x)`

Restituisce `true` se l'importo rappresentato dall'oggetto che esegue il metodo è minore di quello specificato tramite l'argomento, `false` in caso contrario.

Supponete di disporre di uno sconto percentuale applicabile a un prodotto a scelta in un insieme di prodotti acquistati. Chiaramente, converrà scegliere di applicare lo sconto al prodotto più costoso.

Scrivete un'applicazione che legga i prezzi dei prodotti acquistati e la percentuale di sconto, e comunichi il prezzo totale da pagare, avendo applicato lo sconto al prodotto più costoso.

Esempio:

```
Prezzo: euro? 4
        cent? 33
Altri prodotti? s
Prezzo: euro? 10
        cent? 60
Altri prodotti? s
Prezzo: euro? 5
        cent? 12
Altri prodotti? n
Sconto? 5
TOTALE EURO 20,05
SCONTO 5% su 10,60: EURO 0,53
TOTALE SCONTATO: EURO 19,52 (Lire 37796)
```

Suggerimento

Costruite prima di tutto un'applicazione che legga una sequenza di importi e ne calcoli la somma. Modificate poi l'applicazione in modo che, durante la lettura della sequenza, determini anche l'importo maggiore tra quelli letti. Infine, ispirandovi agli esercizi precedenti, modificate l'applicazione in modo che effettui quanto richiesto.

3.27 Scrivete un'applicazione che legga un prezzo espresso in euro (euro e centesimi di euro) e comunichi:

- il prezzo letto scritto prima in cifre e poi in lettere (nel formato che si utilizza per gli assegni: ad esempio 12,05 si scrive dodici/05);
- l'equivalente in lire scritto in cifre e in lettere.

L'applicazione deve poi leggere una percentuale di sconto (come numero intero) da applicare al prezzo e deve poi comunicare:

- lo sconto in euro;
- il prezzo scontato in euro.

Infine, l'applicazione deve leggere un importo pagato espresso in euro e comunicare il debito o il resto rispetto al prezzo scontato (utilizzate le classi `Importo` e `Intero` del package `prog.utili`).

Esempi di esecuzione:

Inserire il prezzo:

--- euro? 12

--- centesimi? 5

Prezzo: EURO 12,05 (dodici/05)

Prezzo: Lire 23332 (ventitremilatrecentotrentadue)

Percentuale di sconto? 10

Sconto: EURO 1,20

Prezzo scontato: EURO 10,85

Inserire l'importo pagato:

--- euro? 100

--- centesimi? 0

Importo pagato: EURO 100,00

Resto: EURO 89,15

Inserire il prezzo:

--- euro? 25

--- centesimi? 40

Prezzo: EURO 25,40 (venticinque/40)

Prezzo: Lire 49181 (quarantanovemilacentoottantuno)

Percentuale di sconto? 2

Sconto: EURO 0,50

Prezzo scontato: EURO 24,90

Inserire l'importo pagato:

--- euro? 20

--- centesimi? 0

Importo pagato: EURO 20,00

Debito: EURO 4,90

- 3.28 Il package `java.util` fornisce una classe `Random` le cui istanze sono generatori di numeri con distribuzione casuale. La classe dispone di un costruttore privo di argomenti che crea un generatore. Tra i metodi forniti vi sono:

- `public int nextInt()`
`public boolean nextBoolean()`

Restituiscono, rispettivamente, un valore `int` e un valore `boolean` generati a caso.

- `public int nextInt(int n)`

Restituisce un valore intero, generato a caso, maggiore o uguale a zero e minore del valore specificato tramite l'argomento.

Scrivete un'applicazione che generi e visualizzi una sequenza di 20 numeri interi qualsiasi generati casualmente. L'applicazione dovrà poi visualizzare la loro somma.

3.29 Modificate l'applicazione scritta per l'esercizio precedente in modo che richieda all'utente quanti numeri dovranno essere generati e un numero x . L'applicazione dovrà generare numeri non negativi minori di x .

3.30 Servendovi della classe `Intero`, modificate l'applicazione scritta per l'esercizio precedente in modo che visualizzi i numeri generati e la loro somma in lettere anziché in cifre.

3.31 Utilizzando il metodo `nextBoolean` della classe `Random`, scrivete un'applicazione che simuli una sequenza di lanci di moneta, dove il numero di lanci viene inserito preliminarmente dall'utente. Calcolate via via la percentuale di lanci che forniscono come risultato "testa" e la percentuale di lanci che forniscono come risultato "croce". Al crescere del numero di lanci le due percentuali dovrebbero stabilizzarsi a intorno al 50%.

Tipi primitivi e tipi enumerativi

Nel capitolo precedente abbiamo trattato le strutture di controllo e il tipo `boolean`, fondamentale per esprimere le condizioni. In questo capitolo studieremo in modo sistematico gli altri tipi primitivi di Java e le espressioni. Infine presenteremo i tipi enumerativi, che sono stati introdotti in Java a partire dalla versione 5.

4.1 Espressioni

Finora abbiamo utilizzato il termine “espressione” senza darne una definizione precisa. In questo paragrafo, dopo aver descritto che cosa si intende per espressione nel linguaggio Java, faremo alcune considerazioni relative alle espressioni coi tipi `int` e `boolean`.

Un’espressione è una porzione di codice Java, che ha un *tipo* e un *valore*. Ad esempio, se `i` e `j` sono di tipo `int`, l’espressione

`i + j`

ha tipo `int`, mentre l’espressione

`"ciao".toUpperCase()`

ha tipo `String`, in quanto il suo risultato è un riferimento a un oggetto `String`. Poiché il linguaggio è fortemente tipizzato, il tipo di un’espressione può essere desunto dal codice del programma. Al contrario, il valore dipende dalla particolare esecuzione.¹

Le espressioni più semplici sono i *letterali*, cioè espressioni che vengono interpretate letteralmente, così come sono scritte. Ad esempio `25` e `"pippo"` sono due letterali. Il nome di una variabile è un’espressione il cui valore è il contenuto della variabile stessa. L’invocazione di un metodo è un’espressione il cui valore è il risultato restituito dal metodo. Ad esempio, l’invocazione del metodo `equals` della classe `Frazione` è un’espressione che fornisce un risultato di tipo `boolean`.

¹ Esistono linguaggi di programmazione in cui il tipo di una variabile può cambiare durante l’esecuzione del programma; di conseguenza una stessa espressione, valutata in due momenti diversi, può produrre risultati di tipi differenti.

L'espressione `i + j` è ottenuta combinando tra loro espressioni elementari, cioè gli identificatori delle due variabili. In generale le espressioni di tipo `int` possono essere combinate tra loro tramite gli *operatori binari* `+`, `-`, `*`, `/` e `%`, che denotano rispettivamente le operazioni di somma, sottrazione, prodotto, divisione e resto della divisione. Il risultato di queste operazioni è sempre di tipo `int` (vedremo che, grazie all'overloading degli operatori, gli stessi simboli saranno utilizzati per indicare le analoghe operazioni con altri tipi numerici). Tra questi operatori valgono le solite regole di precedenza stabilite dall'algebra. Per modificare le precedenze è possibile raggruppare parti di un'espressione mediante parentesi tonde, come in:

$$4 * (i + j) - (3 + k * (i - j))$$

dove le variabili `k`, `i` e `j` siano state dichiarate di tipo `int`.

In Java, come del resto nel linguaggio C, l'assegnamento denotato dall'operatore `=` è un'espressione il cui tipo è quello della variabile alla quale viene assegnato il valore e il cui risultato è il valore assegnato. Ad esempio, se la variabile `x` è stata dichiarata di tipo `int`, l'espressione:

$$x = 10$$

è di tipo `int` e ha come risultato 10. Questa espressione potrebbe essere utilizzata in un'altra espressione, ad esempio a destra di un altro assegnamento. Se `x` e `y` sono state dichiarate di tipo `int`, l'esecuzione dell'istruzione:

$$y = x = 10;$$

avviene come segue: si calcola, come sempre, il valore dell'espressione scritta a destra del simbolo di assegnamento, cioè l'espressione `x = 10`. Questa espressione viene valutata a sua volta come un normale assegnamento, ponendo quindi il valore 10 nella variabile `x`, e producendo il risultato 10, che può dunque essere assegnato a `y`. Consideriamo ora il seguente frammento di codice:

```
int i, j;
i = 3;
j = i + (i = 5);
```

Dopo avere assegnato alla variabile `i` il valore 3, si deve eseguire l'istruzione di assegnamento a `j`. Per il calcolo dell'espressione a destra del simbolo di assegnamento occorre sommare al valore contenuto in `i` (3) il risultato dell'espressione `i = 5`. Questa espressione ha risultato 5 e, come *effetto collaterale*, modifica il contenuto della variabile `i`. Pertanto, dopo l'esecuzione di questa porzione di codice, le variabili `i` e `j` conterranno, rispettivamente, i valori 5 e 8. Consideriamo ora il frammento:

```
int i, j;
i = 3;
j = (i = 5) + i;
```

Analizzando i singoli passi necessari per la sua esecuzione, come nel caso precedente, possiamo osservare che le variabili *i* e *j* alla fine conterranno, rispettivamente, i valori 5 e 10. Come si può notare, l'uso di espressioni con effetti collaterali rende i programmi più difficili da leggere. Pertanto è bene evitarne l'impiego, o farne un uso molto prudente.

Sul tipo *int* sono disponibili gli operatori unari di *incremento* e di *decremento*, indicati rispettivamente con i simboli ***++*** e ***--***. Ambedue le istruzioni:

x++;

e

++x;

incrementano il valore contenuto nella variabile *x*. Come gli assegnamenti, anche ***x++*** e ***++x*** sono espressioni con effetti collaterali, che possono essere utilizzate all'interno di espressioni complesse. Ambedue le espressioni hanno come effetto collaterale l'incremento della variabile *x*. Il valore dell'espressione ***x++*** è il contenuto di *x* *prima* dell'incremento (leggendo l'espressione da sinistra a destra possiamo immaginare che prima si fornisca il valore di *x* e poi si incrementi la variabile); il valore di ***++x*** è invece il contenuto di *x* *dopo* l'incremento (anche in questo caso, leggendo l'espressione da sinistra a destra, troviamo prima il simbolo di incremento e poi il nome della variabile; dunque, prima si esegue l'incremento e poi si fornisce il valore contenuto in *x*). Ad esempio, dopo l'esecuzione del frammento di codice:

```
int x = 3;
int y = x++;
```

le variabili *x* e *y* contengono rispettivamente 4 e 3. Dopo l'esecuzione di:

```
int x = 3;
int y = ++x;
```

le due variabili contengono entrambe il valore 4.

Presentiamo ora ulteriori esempi contenenti effetti collaterali. Per prima analizziamo l'espressione seguente, dove *x* e *y* sono come sempre di tipo *int*:

x++ + y + x

L'espressione è valutata da sinistra verso destra. Il primo operando della somma di sinistra è il valore di *x* *prima* dell'incremento; *x* viene incrementata immediatamente dopo la valutazione dell'operando; si valuta poi il secondo operando, cioè *y*. A questo punto è possibile calcolare la somma di sinistra. Il risultato è il primo operando della somma di destra; il secondo operando è il valore presente ora in *x*. Ad esempio, nel caso *x* e *y* contengano prima del calcolo, rispettivamente, i valori 5 e 3, si hanno i seguenti passi.

- Valutazione del primo operando della somma di sinistra (***x++***).

Il valore del primo operando è quello contenuto in *x*, cioè 5. Come effetto collaterale, la valutazione provoca un incremento del valore contenuto in *x*, che diventa 6.

- Valutazione del secondo operando della somma di sinistra (y).
Il valore è 3. Non ci sono effetti collaterali.
- Calcolo della somma di sinistra.
Sommando i due operandi si ottiene 8. Questo risultato è il primo operando della somma di destra.
- Valutazione del secondo operando della somma di destra (x).
Il valore contenuto ora in x è 6. Non ci sono effetti collaterali.
- Valutazione della somma di destra e quindi dell'espressione.
Sommando i valori dei due operandi, 8 e 6, si ottiene il valore dell'espressione, cioè 14. Nel complesso l'espressione ha prodotto come effetto collaterale un incremento del contenuto di x.

Consideriamo ora l'espressione

`++x + y + x;`

Supponendo sempre che x e y contengano rispettivamente 5 e 3, analizziamo i passi d'esecuzione.

- Valutazione del primo operando della somma di sinistra (`++x`).
Poiché viene utilizzato l'operatore `++` in forma prefissa, in questo caso si ha l'effetto collaterale *prima* della valutazione. Dunque il valore di x viene incrementato e diventa 6. Tale valore è anche il valore del primo operando.
- Valutazione del secondo operando della somma di sinistra (y).
Il valore è 3. Non ci sono effetti collaterali.
- Calcolo della somma di sinistra.
Sommando i due operandi si ottiene 9. Questo risultato è il primo operando della somma di destra.
- Valutazione del secondo operando della somma di destra (x).
Il valore contenuto ora in x è 6. Non vi sono effetti collaterali.
- Valutazione della somma di destra e quindi dell'espressione.
Sommando i valori dei due operandi, 9 e 6, si ottiene il valore dell'espressione, cioè 15. Anche in questo caso l'espressione ha prodotto come effetto collaterale un incremento del contenuto di x.

Consideriamo ora il seguente esempio, dove le variabili sono tutte di tipo `int`:

```
if (x++ == --y)
    z = x + y;
else
    z = x - y;
```

Supponendo che tutte le variabili contengano inizialmente 2, vogliamo conoscere il valore che conterranno dopo l'esecuzione del frammento di codice.

La condizione dell'`if` utilizza l'operatore `==`, i cui operandi sono le espressioni `x++` e `--y`. Il calcolo dell'espressione `x++` produce prima il valore, cioè 2, e poi l'effetto collaterale, cioè l'incremento di `x`; l'espressione `--y` produce prima l'effetto collaterale, cioè il decremento di `y`, e poi il valore, cioè 1. Poiché i valori delle due espressioni sono diversi, la condizione è falsa. Viene pertanto eseguito il ramo `else`, in cui si assegna a `z` il risultato della differenza tra `x` e `y`, cioè 2. Pertanto, dopo l'esecuzione del codice, le variabili `x`, `y` e `z` conterranno, rispettivamente, i valori 3, 1 e 2.

Supponiamo ora che i valori contenuti inizialmente nelle variabili `x`, `y` e `z` siano rispettivamente 1, 2 e 0. Gli operandi dell'operatore `==` sono 1 e 1 (il valore di `x` prima dell'incremento e quello di `y` dopo il decremento). Come effetto collaterale il valore di `x` diventa 2 e quello di `y` diventa 1. Poiché la condizione è vera, si esegue l'istruzione immediatamente successiva, il cui effetto è l'assegnamento del valore 3 a `z`.

In un programma capita spesso di eseguire assegnamenti con la stessa variabile a sinistra e all'inizio dell'espressione a destra dell'assegnamento come, ad esempio:

```
x = x + y;
```

con `x` e `y` di tipo `int`. Questa istruzione può essere scritta in forma abbreviata utilizzando l'operatore `+=`, come segue:

```
x += y;
```

Per le altre operazioni sono disponibili operatori simili, indicati con i simboli `-=`, `/=`, `*=` e `%=`.

Molti operatori presentati in questo paragrafo per il tipo `int` sono disponibili, mediante overloading, anche per gli altri tipi numerici che esamineremo in seguito.

Lazy evaluation

Il risultato di un operatore è normalmente determinato dopo i relativi operandi. Ad esempio, date due variabili `x` e `y` di tipo `int`, per calcolare la somma `x + y` è necessario disporre del valore di `x` e di quello di `y`. Questa regola non vale per gli operatori `&&` e `||`, che utilizzano la *lazy evaluation* (valutazione "pigra" o *cortocircuitata*). Se, avendo valutato solo una parte di un'espressione booleana, è già possibile determinare il risultato dell'espressione, la parte che resta non viene valutata.

Si consideri, per esempio, l'espressione:

```
(x > 0) && (y++ != x)
```

dove `x` e `y` sono di tipo `int`. Se `x` contiene un valore minore o uguale a zero, l'intera condizione è sicuramente falsa. Pertanto il secondo operando di `&&`, cioè l'espressione `y++ != x`, non viene valutato. Di conseguenza la variabile `y` non viene in questo caso incrementata. Al contrario, quando `x` contiene un valore positivo, per conoscere il valore dell'intera condizione è necessario

valutare anche l'espressione `y++ != x`. Dunque, in tale circostanza, il valore contenuto in `y` verrà incrementato.

In Java esiste anche una versione degli operatori logici che valuta in maniera completa le espressioni; l'operatore di congiunzione è `&`, mentre quello di disgiunzione è `|`. Ad esempio, in una condizione della forma `a & b`, `b` viene valutata anche quando `a` è falsa.

Consideriamo ad esempio l'espressione precedente in cui si utilizza l'operatore di congiunzione `&`:

```
(x > 0) & (y++ != x)
```

In questo caso, se `x` contiene un valore minore o uguale a zero, anche se l'intera condizione è sicuramente falsa, il secondo operando di `&`, cioè `y++ != x`, viene comunque valutato, e quindi la variabile `y` viene incrementata. A differenza di quanto succedeva nell'espressione con l'operatore `&&`, la valutazione ha gli stessi effetti collaterali indipendentemente dal valore del primo operando.

L'operatore condizionale

Per concludere questa rassegna sulle espressioni descriviamo l'operatore condizionale. Supponiamo di voler assegnare a una variabile `max` il valore più grande tra quelli contenuti in due variabili `x` e `y` (supponiamo che tutte le variabili siano di tipo `int`). In altre parole vogliamo calcolare:

$$\max = \begin{cases} y & \text{se } x < y \\ x & \text{altrimenti} \end{cases}$$

Potremmo utilizzare una selezione della forma:

```
if (x < y)
    max = y;
else
    max = x;
```

Tuttavia, osservando la formula scritta sopra, sarebbe opportuno disporre di un'unica istruzione di assegnamento, in cui l'espressione a destra dell'assegnamento viene selezionata tra due espressioni in base al valore di una condizione: se la condizione `x < y` è vera, si seleziona l'espressione `y`, altrimenti si seleziona l'espressione `x`.

A tale scopo Java fornisce l'*operatore condizionale* `? :`, un operatore ternario (cioè con tre operandi) che permette di effettuare questo tipo di selezione. Nel caso specifico l'espressione da assegnare a `max` può essere scritta come:

```
x < y ? y : x
```

Dunque possiamo scrivere l'assegnamento come:

```
max = x < y ? y : x;
```

Prima del simbolo `?` si scrive la condizione; tra i simboli `?` e `:` si scrive l'espressione da utilizzare nel caso la condizione sia vera; dopo il simbolo `:` si indica l'espressione da utilizzare nel caso la condizione sia falsa.

4.2 Riepilogo degli operatori

Riepiloghiamo brevemente alcuni operatori incontrati finora.

- *Operatori aritmetici.*

Sono gli usuali operatori binari +, -, *, / e %, che trattano i tipi numerici. Esistono anche gli operatori + e - unari, per indicare il segno positivo o negativo.

- *Operatori di incremento e decremento.*

Sono gli operatori ++ e --.

- *Operatori relazionali.*

>	maggiore di
>=	maggiore o uguale a
<	minore di
<=	minore o uguale a
==	uguale a
!=	diverso da

Possono essere applicati a tutti i tipi primitivi. Inoltre gli operatori == e != possono essere applicati anche ai riferimenti (confrontano i riferimenti, non gli oggetti).

- *Operatori logici.*

&	AND logico
	OR logico
^	OR esclusivo
!	negazione logica
&&	AND condizionale (lazy evaluation)
	OR condizionale (lazy evaluation)

- *Operatore condizionale.*

È l'operatore ternario ?:.

- *Operatore di assegnamento.*

È l'operatore =.

- *Operatore di concatenazione.*

È l'operatore +, quando almeno uno dei due operandi sia di tipo String.

- *Operatore di istanziazione.*

È l'operatore new seguito dal nome di un costruttore. Il risultato è un riferimento a un oggetto.

Per stabilire quale tra due operatori debba essere applicato per primo esistono le *regole di precedenza*. Tali regole possono essere reperite nei manuali incertenzi al linguaggio. In caso di dubbio è opportuno utilizzare le parentesi, senza naturalmente abusarne.

A parità di precedenza, le espressioni sono valutate sempre da sinistra verso destra (ad esempio $x + y + z$ equivale a $(x + y) + z$), eccezione fatta per l'operatore $=$, in cui la valutazione va da destra a sinistra ($x = y = z$ equivale a $x = (y = z)$).

In un'espressione gli operandi vengono valutati *prima* dell'operatore, tranne che per gli operatori `&&` e `||` (si veda la trattazione relativa alla *lazy evaluation*), e per l'operatore ternario `? :`, in cui, oltre alla condizione, viene valutata solo l'espressione selezionata.

Esercizi

4.1 Individuate il tipo e il valore delle seguenti espressioni evidenziando eventuali effetti collaterali (si supponga che le variabili x e y siano di tipo `int` e contengano, rispettivamente, 8 e 9):

- (1) $x = y$
- (2) $x == y$
- (3) $x != y$
- (4) $++x$
- (5) $((x - y) > (y = x)) \&\& (x == y--)$
- (6) $((x - y) < (y = x)) \&\& (x == y--)$
- (7) $((x - y) > (y = x)) || (x == y--)$
- (8) $((x - y) < (y = x)) || (x == y--)$
- (9) $!((x - y) > (y = x)) \&\& (x == y--)$
- (10) $!((x - y) < (y = x)) \&\& (x == y--)$
- (11) $!((x - y) > (y = x)) || (x == y--)$
- (12) $!((x - y) < (y = x)) || (x == y--)$
- (13) $x++ == y--$
- (14) $x++ == --y$
- (15) $++x == --y$
- (16) $++x == y--$
- (17) $x == y ? x++ : --y$
- (18) $x != y ? x++ : --y$

4.2 Per ognuno dei seguenti assegnamenti scrivete delle dichiarazioni per le variabili che vi compaiono, così che essi risultino corretti dal punto di vista dei tipi; se ciò non fosse possibile, spicgetne il motivo:

- (1) $x = (y-- > 4) \&\& !z;$

- (2) $x = x == y;$
- (3) $x = x == 1;$
- (4) $x = x = 1;$
- (5) $x = !x;$
- (6) $x = x = x;$
- (7) $x = x == x;$

4.3 Supponete che le variabili *i* e *j* siano state dichiarate di tipo *int*, e che contengano, rispettivamente, i valori 4 e 7. Indicate i valori contenuti nelle due variabili dopo l'esecuzione dei seguenti assegnamenti:

- (1) $j = i + (i = j);$
- (2) $j = i + (i = i);$
- (3) $j = (i = j) + i;$
- (4) $j = (i--) + (--j);$
- (5) $j = i + (j = j + 1) + (j = i + 2) + (i = j);$

4.4 Considerate questa istruzione, in cui le variabili *x* e *y* sono di tipo *int*:

```
if (x != y && (x = y++) > 0)
    x = y;
```

Indicate i valori contenuti nelle variabili *x* e *y* *dopo* l'esecuzione dell'istruzione nei seguenti casi:

- (1) prima dell'esecuzione sia *x* sia *y* contengono 3
- (2) prima dell'esecuzione *x* contiene 2 e *y* contiene 0
- (3) prima dell'esecuzione *x* contiene 2 e *y* contiene 1.

4.5 Considerate la seguente istruzione, in cui le variabili *x* e *y* sono di tipo *int*:

```
if (x <= y && (x = y--) > 0)
    x = y;
```

Indicate i valori contenuti nelle variabili *x* e *y* *dopo* l'esecuzione dell'istruzione nei seguenti casi:

- (1) prima dell'esecuzione sia *x* sia *y* contengono 0
- (2) prima dell'esecuzione *x* contiene 4 e *y* contiene 0
- (3) prima dell'esecuzione *x* contiene 4 e *y* contiene 1.

4.6 Considerate questa istruzione, in cui le variabili `x` e `y` sono di tipo `int`:

```
if (x <= y || (x = y++) > 0)
    x = y;
```

Indicate i valori contenuti nelle variabili `x` e `y` *dopo* l'esecuzione dell'istruzione nei seguenti casi:

- (1) prima dell'esecuzione sia `x` sia `y` contengono 0
- (2) prima dell'esecuzione `x` contiene 3 e `y` contiene 0
- (3) prima dell'esecuzione `x` contiene 3 e `y` contiene 1.

4.7 Considerate questa istruzione, in cui le variabili `x` e `y` sono di tipo `int`:

```
if (x <= y || (x = y--) > 0)
    x = y;
```

Indicate i valori contenuti nelle variabili `x` e `y` *dopo* l'esecuzione dell'istruzione nei seguenti casi:

- (1) prima dell'esecuzione sia `x` sia `y` contengono 0;
- (2) prima dell'esecuzione `x` contiene 4 e `y` contiene 0;
- (3) prima dell'esecuzione `x` contiene 4 e `y` contiene 1.

4.3 Tipi numerici interi

Per la rappresentazione dei numeri interi, oltre a `int`, sono disponibili in Java diversi tipi.

Nella definizione di un linguaggio di programmazione vengono generalmente stabilite le operazioni ammissibili su un certo tipo di dati, ma non viene fissata la rappresentazione dei dati di quel tipo, che dipende invece dalle caratteristiche del processore presente nella macchina su cui avviene l'esecuzione. Pertanto il comportamento del programma può dipendere dalla particolare macchina su cui viene eseguito. Al contrario, i progettisti del linguaggio Java hanno stabilito anche le rappresentazioni dei tipi primitivi. Ad esempio per il tipo `int` si è scelta una rappresentazione a 32 bit, con cui è possibile rappresentare i numeri nell'intervallo che va da -2^{31} a $2^{31} - 1$, cioè da $-2.147.483.648$ a $+2.147.483.647$.² Se un'operazione produce un risultato non rappresentabile in questo range, si ha un *integer overflow*. In questa situazione l'esecuzione del programma prosegue. Tuttavia il risultato non è matematicamente corretto. Infatti, viene ottenuto dalla macchina semplicemente eliminando le cifre che non possono essere rappresentate nei 32 bit. Si consiglia di fare qualche esperimento, provando ad esempio a sommare numeri molto grandi.

² Questi due valori sono disponibili nelle costanti `Integer.MIN_VALUE` e `Integer.MAX_VALUE`, definite nella classe involucro `Integer` descritta più avanti in questo capitolo.

Per trattare numeri interi più grandi si dispone di un tipo primitivo di nome `long`, in cui gli interi sono rappresentati su 64 bit. Le operazioni utilizzate sono le stesse del tipo `int`. Applicate a valori `long`, forniscono *sempre* risultati di tipo `long`.

I letterali interi sono sequenze di cifre, eventualmente precedute dai segni + o -, ed eventualmente seguite dalla lettera L minuscola o maiuscola, utilizzata per identificare i letterali `long`. Ad esempio, la sequenza di caratteri `-3245L` è un letterale `long`, la sequenza `40000` è un letterale `int`.

Oltre ai tipi `int` e `long`, altri due tipi, `byte` e `short`, permettono di rappresentare numeri interi. Ecco un riassunto dei tipi interi e dei valori rappresentabili:

- `byte`, rappresentazione su 8 bit, valori da $-128 (-2^7)$ a $127 (2^7 - 1)$
- `short`, rappresentazione su 16 bit, valori da $-32768 (-2^{15})$ a $32767 (2^{15} - 1)$
- `int`, rappresentazione su 32 bit, valori da $-2147483648 (-2^{31})$ a $2147483647 (2^{31} - 1)$
- `long`, rappresentazione su 64 bit, valori da -2^{63} a $2^{63} - 1$.

4.4 Tipi numerici in virgola mobile

In aggiunta ai numeri interi spesso occorre rappresentare all'interno dei programmi numeri con la virgola. Java fornisce a questo scopo il tipo `double`. Anche per il tipo `double` sono disponibili gli operatori +, -, *, /, che denotano, rispettivamente, le operazioni di somma, sottrazione, moltiplicazione e divisione tra numeri reali. È disponibile anche l'operatore % per il calcolo del resto della divisione (per il significato di questo operatore nel caso del tipo `double` si rimanda alla documentazione relativa al linguaggio). Sono inoltre disponibili gli operatori di incremento e di decremento (++ e --), gli operatori +=, *=, etc., già presentati per il tipo `int`, e gli operatori di confronto.

Tramite il tipo `double` è possibile rappresentare numeri reali compresi in un certo intervallo. In particolare le costanti `Double.MIN_VALUE` e `Double.MAX_VALUE`, della classe involucro `Double`, indicano il più piccolo valore positivo e il più grande valore positivo rappresentabili nel tipo. Si noti che, mentre il tipo `int` è in grado di rappresentare *tutti* i valori interi nell'intervallo da `Integer.MIN_VALUE` a `Integer.MAX_VALUE`, il tipo `double` può rappresentare solo un sottoinsieme dei numeri reali compresi tra `Double.MIN_VALUE` e `Double.MAX_VALUE`. Infatti, poiché la macchina dispone soltanto di una memoria finita, mentre vi sono numeri reali, anche molto piccoli, la cui rappresentazione utilizza un numero infinito di cifre, non tutti i numeri reali sono rappresentabili.

In particolare, nel tipo `double` si utilizza una rappresentazione a 64 bit in *virgola mobile* (*numeri floating point*) o *notazione scientifica*, che permette di rappresentare in poco spazio numeri in valore assoluto molto grandi e molto piccoli. In tale rappresentazione si utilizzano 53 bit per la *mantissa* e i rimanenti 11 per l'*esponente*. Ad esempio, riferendoci alla base 10 cui siamo abituati (anziché alla base 2 utilizzata dalla macchina), osserviamo che il numero `0.000012` può essere scritto come $1.2 \cdot 10^{-5}$. In questa rappresentazione la mantissa è 1.2,

l'esponente è -5 , cioè l'esponente di 10 . Nei programmi questo numero può essere rappresentato dal letterale `double` `1.2E-5`. La lettera E (maiuscola o minuscola) viene utilizzata per separare la mantissa dall'esponente. Esiste anche una rappresentazione dei valori $-\infty$ e $+\infty$.³

Oltre alla forma mantissa ed esponente, come in `1.2E-5` o `5.6e67` o `-5.6e67`, sono letterali `double` tutte le sequenze di cifre (eventualmente precedute dai segni + o -) contenenti un punto decimale, come `3.14`, `2.` e `0.0`. Si noti che, mentre `3` è un letterale di tipo `int`, `3.0` è un letterale di tipo `double`. Inoltre è possibile utilizzare la lettera d (minuscola o maiuscola) alla fine dei letterali per indicare che sono di tipo `double`, come ad esempio `-5.6e67d`, `3.14D`, `2d`.

Nel tipo `double` sono rappresentabili numeri che, in valore assoluto, arrivano a circa 10^{300} . I numeri più prossimi allo zero che si possono rappresentare sono, in valore assoluto, dell'ordine di circa 10^{-300} .

Esiste un altro tipo per i numeri floating point, il tipo `float`. In questo tipo i numeri vengono rappresentati su 32 bit (24 per la mantissa, 8 per l'esponente). I letterali di tipo `float` hanno una struttura simile a quelli di tipo `double`, ma si chiudono *sempre* con una lettera f minuscola o maiuscola, come `-5.6e67f`, `3.14F`, `2f`.

Poiché la rappresentazione in un computer deve necessariamente utilizzare un numero finito di bit, mentre, come già osservato, ci sono numeri reali la cui rappresentazione richiederebbe un numero infinito di bit, nell'aritmetica coi numeri reali si verifica una perdita di precisione. I numeri sono approssimati utilizzando le cifre disponibili. In calcoli successivi e ripetuti, piccole perdite di precisione possono via via accrescere dando risultati molto distanti da quelli corretti. Dello studio di queste problematiche si occupa l'*analisi numerica*.

Esempio: calcolo dell'area del cerchio e della lunghezza della circonferenza

Presentiamo un esempio in cui si utilizzano variabili di tipo `double`: scriviamo una classe contenente un metodo `main` che, dopo avere letto la lunghezza del raggio di un cerchio, calcola e comunica l'area del cerchio e la lunghezza della circonferenza. Per la lettura di valori `double` usiamo il metodo `readDouble` della classe `ConsoleInputManager`.

```
import prog.io.*;
public class CalcoloCerchio {
    public static void main(String[] args) {
        // predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        double raggio, area, circonferenza;
        final double PI = 3.14;
```

³ La rappresentazione utilizzata in Java si basa sullo standard IEEE 754. Maggiori dettagli sono reperibili nei documenti di specifica del linguaggio.

```

//lettura del raggio
raggio = in.readDouble("Inserire la lunghezza del raggio ");

//calcolo di area e circonferenza
area = raggio * raggio * PI;
circonferenza = raggio * 2 * PI;

//comunicazione dei risultati
out.println("Area cerchio = " + area);
out.println("Lunghezza circonferenza = " + circonferenza);
}

}

```

Si osservi che, anziché scrivere direttamente il valore 3.14 nelle espressioni per il calcolo dell'area e della circonferenza, abbiamo utilizzato l'identificatore PI, definito `final double`, con valore 3.14. Il modificatore `final` indica che la variabile PI, una volta inizializzata, non potrà più essere modificata. In altre parole PI è una *costante*. L'uso di costanti rende il programma più leggibile e facilmente modificabile. Ad esempio, volendo utilizzare una stima più accurata del valore di pi greco, ipoteticamente con 4 cifre decimali, è sufficiente modificare il valore assegnato alla costante nella definizione, senza dover modificare tutte le formule che la utilizzano.

Per convenzione, tutte le lettere utilizzate negli identificatori di costante sono maiuscole. Alcune costanti utili sono già definite nelle librerie. Nella libreria Math è definita, ad esempio, proprio una costante PI per il valore di pi greco. Essa può essere utilizzata direttamente nelle classi usando il nome `Math.PI` (la classe Math è definita nel package `java.lang`).

```

import prog.io.*;

class CalcoloCerchio {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        double raggio, area, circonferenza;

        //lettura del raggio
        raggio = in.readDouble("Inserire la lunghezza del raggio ");

        //calcolo di area e circonferenza
        area = raggio * raggio * Math.PI;
        circonferenza = raggio * 2 * Math.PI;
    }
}

```

```

//comunicazione dei risultati
out.println("Area cerchio = " + area);
out.println("Lunghezza circonferenza = " + circonferenza);
}
}

```

Quando in un programma una costante viene utilizzata frequentemente, risulta scomodo indicare ogni volta il nome della classe a cui appartiene. Per questa ragione il linguaggio fornisce un meccanismo di *importazione statica* (*static import*) simile a quello utilizzato per importare le classi.⁴ Ad esempio nel caso dell'applicazione precedente potremmo importare la costante statica PI della classe `java.lang.Math` aggiungendo alle direttive di importazione:

```
import static java.lang.Math.PI;
```

In questo modo, nel corpo dell'applicazione, possiamo utilizzare direttamente la costante PI senza doverle premettere il nome della classe `Math`.

È anche possibile importare tutte le costanti statiche di una classe, ad esempio la direttiva

```
import static java.lang.Math.*;
```

importa tutte le costanti della classe `Math`.⁵

Osserviamo che il meccanismo di importazione statica dovrebbe essere utilizzato con molta parsimonia in quanto riduce la leggibilità dei programmi. In particolare andrebbe utilizzato solo quando nel codice la costante è utilizzata frequentemente. In ogni caso risulta preferibile utilizzare il meccanismo di importazione individuale piuttosto che importare tutte le costanti di una classe.

4.5 Conversioni implicite ed esplicite di tipo

Abbiamo visto che nel linguaggio Java ci sono 4 tipi primitivi per la rappresentazione dei numeri interi e 2 tipi primitivi per la rappresentazione dei numeri in virgola mobile.

Concentriamoci ora sui due tipi principali per i numeri interi, cioè `int` e `long`, e sui tipi in virgola mobile, cioè `float` e `double` (per i tipi `byte` e `short` esistono alcune differenze, che non trattiamo, riscontrabili nei documenti di specifica del linguaggio).

Per questi tipi numerici sono disponibili operatori comuni, come ad esempio l'operatore di somma `+`. In realtà l'operazione denotata da `+` è diversa per ogni tipo di dati: sebbene, matematicamente, si tratti sempre di una somma, l'algoritmo applicato cambia. Per esempio, per sommare due `double` occorre tenere conto della mantissa e dell'esponente; per sommare due `int` è necessario considerare 32 bit; per sommare due `long` bisogna considerarne 64.

⁴ Questo meccanismo è stato introdotto a partire dalla versione 5 del linguaggio.

⁵ In realtà il meccanismo di importazione statica permette di importare, oltre alle costanti, anche altri membri statici di una classe, come i campi statici e i metodi statici illustrati più avanti nel testo.

L'operatore di somma, come gli altri operatori aritmetici, applicato a operandi dello stesso tipo, fornisce un risultato del medesimo tipo. Analizziamo ora che cosa succede quando si scrivono espressioni in cui sono coinvolti tipi differenti.

Supponiamo di avere fornito le seguenti dichiarazioni:

```
int x, y, z;
long i, j, k;
double p, q, r;
```

Gli assegnamenti:

```
z = x + y;
k = i + j;
r = p + q;
```

non presentano problemi perché le variabili di ognuno di essi sono del medesimo tipo. Consideriamo ora gli assegnamenti:

```
k = x + y;
j = i + z;
```

Nel primo assegnamento il lato destro è una somma di due `int`. Pertanto il risultato è di tipo `int`. Tale risultato dev'essere assegnato a una variabile di tipo `long`. A tale scopo, il valore del risultato è convertito (o *promosso*) dalla rappresentazione come `int` alla rappresentazione come `long`. Visto che ogni valore `int` è rappresentabile nel tipo `long`, questa conversione implicita (che prende anche il nome di *coercizione*) può essere effettuata senza problemi e senza perdita d'informazione.

Analizziamo ora il secondo assegnamento. Il lato destro richiede il calcolo dell'operazione indicata con il simbolo `+` tra un operando di tipo `long` e un operando di tipo `int`. Il linguaggio non dispone di un'operazione `+` tra `long` e `int`, ma dispone di un'operazione `+` tra due valori `long` e di un'operazione `+` tra due valori `int`. Anche in questo caso il valore `int`, cioè il valore contenuto nella variabile `z`, è promosso a `long` per il calcolo dell'espressione; si effettua poi una somma tra `long`, il cui risultato viene assegnato a un `long`.⁶ Si osservi che anche in questo modo non c'è perdita di informazione.

Viceversa, si supponga di voler assegnare alla variabile `x` il valore contenuto nella variabile `j`. Potremmo tentare di scrivere:

```
x = j;
```

In questo caso `j`, essendo di tipo `long`, potrebbe contenere un valore non rappresentabile in `x`, che è di tipo `int` (ad esempio tre miliardi). Pertanto assegnamenti di questo genere sono vietati e producono errori di compilazione. Più in generale:

- sono consentite conversioni implicite, cioè promozioni da un tipo a un altro tipo più ampio

⁶ Attenzione: è solo il valore contenuto in `z` (o più precisamente una sua copia) che è promosso a `long` per il calcolo dell'espressione. La variabile `z` e il suo contenuto continuano a essere di tipo `int`.

- non sono consentite conversioni implicite verso tipi più ristretti.

Per trasformare la rappresentazione di un valore da un tipo a un tipo più ristretto, è necessario indicarlo nel programma *in maniera esplicita*. Per questo si usa la *forzatura o cast*. Per effettuare un cast, si fa precedere l'espressione da trasformare dall'*operatore di cast*, costituito dal nome del tipo al quale si vuole forzare l'espressione scritto tra parentesi tonde, seguito dall'espressione:

(nome_di_tipo) espressione

Ad esempio, per assegnare alla variabile *x* il contenuto della variabile *j* occorre forzare la rappresentazione del valore di *j* al tipo *int*. L'assegnamento può quindi essere scritto come:

x = (int) j;

Se il valore contenuto in *j* si trova nell'intervallo dei valori rappresentabili nel tipo *int* non ci sono problemi. Se, al contrario, il valore in *j* non è rappresentabile nel tipo *int*, il cast provocherà *perdita d'informazione*: il valore assegnato a *x* sarà ottenuto considerando una parte della rappresentazione binaria del valore in *j*. L'obbligo di utilizzare cast esplicativi quando è possibile una perdita d'informazione riduce la possibilità d'errore. Introducendo un cast, il programmatore dichiara di essere cosciente di utilizzare un'operazione che può provocare perdita d'informazione.

L'operatore di *cast* è uno degli operatori a massima precedenza. Ad esempio, in

x = (int) j + k;

l'operatore *(int)* ha la precedenza rispetto all'operatore *+*. Dunque, in questo caso, *+* è applicato al risultato di *(int) j* (che è di tipo *int*) e a *k* (che è di tipo *long*). Quindi il risultato di *(int) j* dovrebbe essere promosso a *long* prima del calcolo della somma, che fornirebbe risultato *long*, non assegnabile a *x*, che è *int*. Invece possiamo forzare ambedue gli operandi a *int* ed effettuare una somma tra *int* scrivendo:

x = (int) j + (int) k;

oppure sommare valori *long* e forzare il risultato a *int* scrivendo:

x = (int) (j + k);

Gli esempi appena presentati riguardavano i tipi interi. Possiamo estendere la trattazione e includere anche i tipi in virgola mobile. Sono infatti possibili conversioni implicite anche da tipi interi a tipi floating point e da *float* a *double*. In questi casi, tuttavia, ci può essere *perdita di precisione*. Ad esempio, non tutti gli *int* sono rappresentati nel tipo *float*: nella conversione un *int* non rappresentabile sarà approssimato dal *float* più vicino.

Per esemplificare, consideriamo il seguente codice:

```
import prog.io.*;
class Precisione {
```

```

public static void main(String[] args) {
    //predisposizione dei canali di comunicazione
    ConsoleOutputManager out = new ConsoleOutputManager();

    int x = 2109876543;
    float y = x;
    int z = (int) y;

    out.println("x = " + x);
    out.println("y = " + y);
    out.println("z = " + z);
}
}

```

in cui il valore dell'intero contenuto in `x` è convertito a `float` e poi di nuovo a `int`. A causa della perdita di precisione, il valore `int` ottenuto alla fine è differente da quello iniziale. In particolare viene prodotto il seguente output:

```

x = 2109876543
y = 2.10987648E9
z = 2109876480

```

Come nei casi precedenti, le conversioni in senso inverso devono essere esplicitate mediante cast.

Esempio: calcolo della media

Vogliamo ora scrivere un programma che legga tre numeri interi e ne calcoli la media. Per questi esempi iniziali ci serviamo, come sempre, di un'unica classe con un metodo `main`. Utilizzando lo schema degli esempi sviluppati in precedenza, possiamo facilmente scrivere questo codice:

```

import prog.io.*;

class Media {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int x, y, z, media;

        x = in.readInt("Inserisci il primo numero ");
        y = in.readInt("Inserisci il secondo numero ");

```

```

z = in.readInt("Inserisci il terzo numero ");

media = (x + y + z) / 3;

out.println(media);
}
}

```

Dal punto di vista logico il programma è non corretto, perché la media di tre numeri interi può essere un numero con la virgola. Riscriviamo il codice dichiarando `media` di tipo `double`:

```

import prog.io.*;

class Media {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int x, y, z;
        double media;

        x = in.readInt("Inserisci il primo numero ");
        y = in.readInt("Inserisci il secondo numero ");
        z = in.readInt("Inserisci il terzo numero ");

        media = (x + y + z) / 3;

        out.println(media);
    }
}

```

Anche in questo caso il programma è scorretto. Fornendo, ad esempio, come dati i numeri 8, 2 e 3, il risultato del programma sarà, sorprendentemente, 4, senza alcuna cifra decimale, nonostante la variabile `media` sia stata dichiarata di tipo `double`.

Analizziamo l'assegnamento alla variabile `media`. L'espressione scritta a destra richiede il calcolo di una divisione, indicata dall'operatore `/`. Grazie all'overloading, il simbolo `/` può indicare tipi di divisione diversi: in particolare la divisione intera tra `int` o la divisione in virgola mobile tra `double`. Per comprendere quale divisione sarà applicata, dobbiamo *osservare esclusivamente il tipo degli operandi*, senza considerare l'utilizzo del risultato. In questo caso ambedue gli operandi, cioè l'espressione `(x + y + z)` e il letterale 3, sono di tipo `int`. Pertanto `/ indica la divisione intera`. Il valore `int` ottenuto come risultato è convertito nella rappresentazione `double` solo alla fine, per realizzare l'assegnamento alla variabile `media`.

Per far sì che venga effettuata una divisione `double`, che produrrà un risultato con i decimali, è necessario che almeno uno dei due operandi sia di tipo `double`. La soluzione più semplice consiste nell'usare come secondo operando il letterale `double` 3.0:

```
import prog.io.*;
class Media {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int x, y, z;
        double media;

        x = in.readInt("Inserisci il primo numero ");
        y = in.readInt("Inserisci il secondo numero ");
        z = in.readInt("Inserisci il terzo numero ");

        media = (x + y + z) / 3.0;

        out.println(media);
    }
}
```

In alternativa, si può forzare il primo operando a `double` scrivendo:

```
media = (double) (x + y + z) / 3;
```

Si osservi che invece l'assegnamento:

```
media = (double) ((x + y + z) / 3);
```

non produrrebbe l'effetto desiderato.

Conversioni implicite al tipo String

In aggiunta alla somma di numeri, abbiamo già utilizzato l'operatore `+` per indicare la concatenazione di stringhe. Talvolta abbiamo impiegato come operandi di `+` una stringa e un valore numerico. In questi casi, come abbiamo osservato, il valore numerico è sempre convertito in stringa, e `+` indica la concatenazione di stringhe. Per capire quale operazione sarà eseguita occorre in ogni caso prestare molta attenzione agli operandi di `+`. Si supponga, ad esempio, di avere memorizzato in due variabili `i` e `j` di tipo `int` due valori interi letti da tastiera e di volerne visualizzare la somma.

Si potrebbe scrivere:

```
out.print("La somma vale ");
out.println(i + j);
```

oppure:

```
out.println("La somma vale " + i + j);
```

o, ancora:

```
out.println("La somma vale " + (i + j));
```

Sebbene sembrino molto simili, queste soluzioni non danno tutte lo stesso risultato. Una di esse, in particolare, non dà il valore della somma.

In generale un valore di un qualsiasi tipo (sia esso primitivo o riferimento) viene automaticamente convertito a stringa quando appare come operando di + e l'altro operando è di tipo String:

- se il valore è di un tipo primitivo, allora viene prodotta una stringa corrispondente al valore;
- se il valore è di un tipo riferimento, allora viene prodotta una stringa richiamando il metodo `toString` dell'oggetto riferito.

Si supponga, ad esempio, di avere fatto le seguenti dichiarazioni:

```
String s, t;
Frazione f, g;
```

e si supponga che le variabili s f e g contengano, in istruzioni successive, riferimenti a oggetti. L'esecuzione dell'istruzione

```
t = s + g;
```

è equivalente all'esecuzione di

```
t = s + g.toString();
```

Infatti l'operatore + indica una concatenazione tra stringhe, perché uno dei due operandi è una stringa. Questo implica che anche l'altro operando sia convertito automaticamente a stringa. Ciò avviene, nel caso dei riferimenti, invocando il metodo `toString`.

Al contrario, l'istruzione

```
t = f + g;
```

non ha alcun senso, in quanto l'operatore + non è definito tra due riferimenti di tipo Frazione.

Come nel caso del metodo `equals`, ogni classe è dotata di un metodo `toString`, anche se non esplicitamente definito dal programmatore.

4.6 Il tipo char

Il tipo primitivo `char` rappresenta i caratteri Unicode, una codifica dell'alfabeto internazionale che consente di rappresentare, oltre ai caratteri dell'alfabeto italiano e inglese, anche i caratteri della maggior parte delle lingue. I letterali di tipo `char` sono rappresentati racchiudendo fra apici singoli ('') un carattere oppure una *sequenza di escape*. Le sequenze di escape sono sequenze di caratteri che iniziano con il carattere speciale \. Ad esempio i caratteri dell'alfabeto Unicode possono essere rappresentati da sequenze di escape della forma \uXXXX, dove XXXX è un numero esadecimale, cioè un numero le cui cifre possono assumere i valori da 0 a 9 e da A a F (oppure da a a f). Ad esempio, la lettera A è rappresentata dalla sequenza \u0041, e quindi la definizione

```
char c = '\u0041';
```

dichiara `c` come una variabile di tipo `char` e le assegna il valore \u0041, che rappresenta appunto la lettera A. Si può ottenere lo stesso effetto con la dichiarazione

```
char c = 'A';
```

in cui è utilizzato direttamente il carattere 'A'. Alcune sequenze di escape possono essere impiegate per rappresentare caratteri speciali, come '\t' per il carattere di tabulazione, '\n' per il ritorno a capo, '\'' per l'apice e '\\\' per il carattere \.

In Java anche il tipo `char` è un tipo intero. Pertanto è possibile utilizzare per i `char` tutte le operazioni disponibili per i tipi interi. Infatti i `char` sono rappresentati su 16 bit, e possono essere visti come interi senza segno, con valori che vanno da zero ('\u0000') a 65535 ('\uffff'). Il valore numerico corrispondente a un carattere non è altro che il suo codice nella tabella Unicode. Se, ad esempio, una variabile `c` è dichiarata di tipo `char`, l'effetto dell'istruzione

```
c++;
```

è quello di incrementare il contenuto di `c` di 1, cioè di sostituire il carattere contenuto in `c` con il carattere immediatamente successivo nella codifica Unicode. Se, ad esempio, `c` contiene il carattere 'A', dopo l'esecuzione di `c++` conterrà il carattere 'B'.

È possibile utilizzare i soliti operatori di confronto, come `>`, `>=`, `==`, etc. I caratteri sono confrontati in base alla loro codifica Unicode.

Esempio: calcolo del numero di occorrenze di ciascuna vocale in una stringa

Vogliamo scrivere un'applicazione che legga una stringa di caratteri da tastiera, calcoli e comunichi all'utente quante volte ciascuna vocale appare nella stringa.

A tal fine sarà utile disporre di 5 contatori, uno per ciascuna vocale, inizializzati a zero:

```
int na = 0, ne = 0, ni = 0, no = 0, nu = 0;
```

Dopo avere letto la stringa e averne memorizzato il riferimento in una variabile `s`, possiamo esaminarla per determinare il numero di occorrenze di ciascuna vocale.

Ricordiamo che in una stringa i caratteri sono numerati a partire dalla posizione 0. Pertanto è sufficiente far scorrere i caratteri della stringa, dalla posizione 0 alla posizione `s.length() - 1`, esaminandoli uno alla volta. Utilizzeremo pertanto un ciclo della forma:

```
for (int i = 0; i < s.length(); i++)
    controlla il carattere in posizione i
```

Per ottenere un singolo carattere di una stringa, possiamo chiedere alla stringa stessa di fornircelo usando il seguente metodo della classe `String`:

- `public char charAt (int indice)`

Restituisce il carattere nella posizione specificata. Gli indici ammissibili sono compresi fra 0 e `length() - 1`. Il primo carattere della stringa ha indice 0, il secondo ha indice 1, e così via. Se `indice` ha un valore negativo o maggiore o uguale a `length`, il metodo dà luogo a un errore in fase di esecuzione.

Utilizzando questo metodo possiamo riscrivere il ciclo come:

```
char c;
for (int i = 0; i < s.length(); i = i + 1) {
    c = s.charAt(i);
    if (c contiene una vocale)
        incrementa il contatore corrispondente;
}
```

La selezione dovrà essere articolata in una serie di casi, uno per ogni vocale. Inoltre, nelle condizioni dovremo tenere conto del fatto che la lettera può essere minuscola o maiuscola. Una possibile codifica della selezione è la seguente:

```
if (c == 'a' || c == 'A')
    na++;
else if (c == 'e' || c == 'E')
    ne++;
else if (c == 'i' || c == 'I')
    ni++;
else if (c == 'o' || c == 'O')
    no++;
else if (c == 'u' || c == 'U')
    nu++;
```

Infine, dopo il ciclo ci sarà una semplice fase di comunicazione del risultato.

Ecco il testo completo di un'applicazione `OccorrenzeVocali` il cui metodo `main` è costruito secondo le linee indicate sopra:

```
import prog.io.*;
```

```

class OccorrenzeVocali {

    public static void main(String [] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        String s = in.readLine("Inserire la stringa da esaminare ");
        int na = 0, ne = 0, ni = 0, no = 0, nu = 0;
        char c;

        for (int i = 0; i < s.length(); i = i + 1) {
            c = s.charAt(i);
            if (c == 'a' || c == 'A')
                na++;
            else if (c == 'e' || c == 'E')
                ne++;
            else if (c == 'i' || c == 'I')
                ni++;
            else if (c == 'o' || c == 'O')
                no++;
            else if (c == 'u' || c == 'U')
                nu++;
        }

        out.println("Numero di occorrenze della vocale a: " + na);
        out.println("Numero di occorrenze della vocale e: " + ne);
        out.println("Numero di occorrenze della vocale i: " + ni);
        out.println("Numero di occorrenze della vocale o: " + no);
        out.println("Numero di occorrenze della vocale u: " + nu);
    }
}

```

Esercizi

4.8 Supponete che siano state fornite le seguenti dichiarazioni:

```

int x, y, z;
long i, j, k;
double p, q, r;

```

Stabilite se ognuna delle seguenti espressioni è costruita correttamente. In caso affermativo, indicate il tipo del risultato evidenziando eventuali conversioni implicite, perdite d'informazione ed effetti collaterali:

- (1) `i = j`
- (2) `i == j`
- (3) `i = x`
- (4) `x = i`
- (5) `i == x`
- (6) `x == i`
- (7) `p + i + j`
- (8) `p + (i + j)`
- (9) `p = x / y`
- (10) `p = i / y`
- (11) `p = q / y`
- (12) `p = (double) i / y`
- (13) `p = i / (long) y`
- (14) `x = (int) p`
- (15) `x = (int) p + q`
- (16) `x = (int) (p + q)`
- (17) `x = (int) p + y`
- (18) `x = (int) (p + x)`
- (19) `i = (int) i`
- (20) `x + 10`
- (21) `x + 10L`
- (22) `x + 10.0`
- (23) `x + 10F`
- (24) `x + 1e1`
- (25) `x + (int) 1e1`
- (26) `y = x + 10`
- (27) `y = 10L`
- (28) `r = x + 10`
- (29) `r = x + 10L`
- (30) `r = x + 10.0`

4.9

4.10

4.11

- (31) `r = x + 10F`
- (32) `r = x + 1e1`
- (33) `r = x + (int) 1e1`

4.9 Per ognuno dei seguenti assegnamenti scrivet le dichiarazioni per le variabili che vi compaiono, in modo che essi risultino corretti dal punto di vista dei tipi; se ciò non fosse possibile, spiegatene il motivo.

- (1) `x += 3;`
- (2) `x = y / 3.0;`
- (3) `x = x + x;`
- (4) `x = x + x;`
- (5) `x = x + x;`
- (6) `x = (double) (y + z);`
- (7) `x = 4 * 3L;`
- (8) `x = 'x';`

4.10 Siano `c` e `d` due variabili di tipo `char`. Esprimete in linguaggio Java le seguenti condizioni:

- (1) `c` e `d` contengono lo stesso carattere;
- (2) `c` e `d` contengono caratteri differenti;
- (3) nella codifica Unicode il carattere contenuto in `c` precede quello contenuto in `d`;
- (4) nella codifica Unicode il carattere contenuto in `c` precede immediatamente quello contenuto in `d`.

4.11 Dopo avere fornito le opportune dichiarazioni di variabile, riscrivete ognuno dei tre frammenti di codice che seguono sostituendo i cicli `for` con cicli `while`.

- (1) `for (i = i++; i <= 'Z'; i++)
 out.println(i);`
- (2) `for (i = i - 1; i >= j; i--)
 out.println(i);`
- (3) `for (i = j; i > 0; i /= 2)
 out.println(i);`
- (4) `for (i = 1; i <= 10; i += 2)
 out.println(i);
 x = x + 2;`
- (5) `for (i = 1; i <= 10; i += 2) {
 out.println(i);
 x = x + 2;
}`

```
(6) for (i = 1; i <= 10; i += 2) {
    out.println(i);
    x = x + 2;
}

(7) for (i = 1; i <= 10; i += 2) {
    out.println(i);
    x = x + 2;
```

- 4.12 Per ognuno dei seguenti frammenti di codice individuate opportune dichiarazioni di variabile. Specificate inoltre, in funzione dei valori iniziali delle variabili, quante volte sarà eseguita l'istruzione indicata con il commento `/*`. Riscrivete infine il frammento sostituendo i cicli `for` con cicli `while`.

```
(1) for (i = 0; i <= n; i++)
    for (j = m; j >= n; j--)
        out.println(i + "" + j); /*

(2) for (c = 'A'; c <= 'Z'; c++)
    for (j = 'z'; j >= 'a'; j--)
        out.println(c + j);

(3) for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j++)
        out.println(i + "" + j); /*

(4) for (i = -m; i <= n; i++)
    for (j = -10; j <= 10; j++)
        out.println(i + "" + j); /*

(5) for (i = 1; i <= n; i++)
    for (j = n; j >= 1; j--)
        for (k = -n; k <= n; k++)
            out.println(" " + i + j + k); /*
```

- 4.13 Supponete di avere tre variabili di tipo `int`: `x`, `y` e `z`. Scrivete l'output prodotto da ognuna delle seguenti istruzioni nell'ipotesi che le tre variabili contengano, rispettivamente, i valori 1, 2 e 3:

```
(1) out.println(x + y + z);
(2) out.println(x + y + z + " ");
(3) out.println(x + y + " " + z);
(4) out.println(" " + x + y + z);
(5) out.println("x" + y + z);
(6) out.println(x + y + "z");
(7) out.println("x + y" + z);
```

```
(8) out.println(x + (y + "z"));
(9) out.println("x" + (y + z));
```

4.14 Scrivete un'applicazione che, lette due stringhe inserite dall'utente utilizzando la tastiera, visualizzi a video le seguenti informazioni:

- (1) il numero totale di caratteri presenti nelle due stringhe;
- (2) la stringa più lunga tra le due lette;
- (3) il numero complessivo di lettere A (maiuscole e minuscole) presenti nella prima stringa letta.

Se ad esempio vengono inserite le stringhe Buona giornata! e Ciao, come stai?, l'output prodotto dovrà essere:

```
31
Ciao, come stai?
3
```

4.15 Scrivete un'applicazione che, letta una stringa inserita dall'utente utilizzando la tastiera, visualizzi a video:

- (1) la stringa letta, scritta tutta in caratteri maiuscoli, sottolineata (cioè seguita, sulla linea successiva del video, da una sequenza di caratteri ~ pari alla sua lunghezza; si supponga che la stringa letta sia lunga al massimo 20 caratteri);
- (2) il numero complessivo di lettere minuscole presenti nella stringa letta.

Se, ad esempio, viene inserita la stringa 30 Ottobre 2003, l'output prodotto dovrà essere:

```
30 OTTOBRE 2003
-----
6
```

4.16 Supponete che siano state fornite le seguenti dichiarazioni:

```
long i, j, k;
boolean b, c, d;
```

Stabilite se ognuna delle seguenti espressioni è costruita correttamente. In caso affermativo, indicate il tipo del risultato evidenziando eventuali conversioni implicite, perdite d'informazione ed effetti collaterali:

- (1) b = i == j
- (2) b = i = j

- (3) $b == (i == j)$
- (4) $b = c = d$
- (5) $b = c == d$
- (6) $b == (c == d)$

4.7 I tipi enumerativi

I tipi enumerativi presenti in molti linguaggi di programmazione tra cui Java (a partire dalla versione 5.0) vengono definiti elencando i possibili valori, detti *costanti del tipo enumerativo*. Come vedremo nel Capitolo 8, in Java i tipi enumerativi vengono definiti e documentati in modo simile alle classi. Consideriamo ad esempio il tipo enumerativo `MeseDellAnno` definito nel package `prog.utili`. I suoi possibili valori sono, nell'ordine:

GENNAIO FEBBRAIO MARZO APRILE MAGGIO GIUGNO LUGLIO AGOSTO SETTEMBRE
OTTOBRE NOVEMBRE DICEMBRE

Una variabile di tipo enumerativo viene dichiarata nel modo usuale, scrivendo il nome del tipo seguito dal nome della variabile; ad esempio l'istruzione

```
MeseDellAnno mese;
```

dichiara `mese` come una variabile di tipo `MeseDellAnno`. A tale variabile è possibile assegnare valori specificati dalle costanti del tipo enumerativo `MeseDellAnno`. È possibile accedere ai valori di un tipo enumerativo utilizzando la sintassi

tipo_enumerativo.valore

Ad esempio l'assegnamento

```
mese = MeseDellAnno.APRILE
```

assegna alla variabile `mese` il valore `APRILE`. Gli enumerativi sono particolari classi che, come vedremo nella parte relativa all'implementazione, si aprono con la parola chiave `enum` al posto di `class`. I valori di tipo enumerativo sono a tutti gli effetti dei riferimenti a degli oggetti e le variabili di tipo enumerativo sono a tutti gli effetti variabili di tipo riferimento. Quindi è possibile assegnare a una variabile di tipo enumerativo anche il letterale `null`. Mentre una classe normalmente può essere istanziata un numero arbitrario di volte, invocandone il costruttore, per un tipo enumerativo l'insieme delle istanze è fissato: vi è un solo oggetto per ciascuna costante. Tale oggetto viene costruito automaticamente dalla Java Virtual Machine. Pertanto non è possibile invocare il costruttore di un tipo enumerativo. L'utente può esclusivamente utilizzare i riferimenti rappresentati dai valori del tipo enumerativo e non può modificarli.

Come tutti gli oggetti, anche i valori di un tipo enumerativo mettono a disposizione dei comportamenti; ad esempio, come tutti gli oggetti, forniscono un metodo `toString`.

Due metodi di cui tutti i valori di tipo enumerativo dispongono sono:

- **public String name()**

Restituisce il nome della costante enumerativa, ad esempio se invocato tramite la costante APRILE del tipo MeseDellAnno restituisce la stringa "APRILE".

- **public int ordinal()**

Restituisce il numero ordinale del valore enumerativo che esegue il metodo, cioè restituisce il numero che identifica la posizione del valore enumerativo all'interno della sequenza dei valori previsti per il tipo enumerativo. Il primo valore del tipo enumerativo ha convenzionalmente ordinale 0. Ad esempio, nel tipo enumerativo MeseDellAnno, i valori compaiono ordinati da GENNAIO a DICEMBRE nel modo usuale, l'ordinale di GENNAIO è 0, quello di DICEMBRE è 11.

Vedremo in seguito come l'ordine dei valori di un tipo enumerativo abbia un ruolo importante nel loro utilizzo. Per ora ci limitiamo a riportare un semplice esempio di utilizzo dei metodi introdotti. Le istruzioni:

```
MeseDellAnno mese = MeseDellAnno.APRILE;
out.println(mese.toString());
out.println(mese.ordinal());
out.println(mese.name());
```

hanno come effetto quello di visualizzare:

```
Aprile
3
APRILE
```

Si osservi la differenza fra la prima stringa prodotta e l'ultima. La prima, restituita dal metodo `toString`, è stata scelta dall'implementatore della classe, mentre l'ultima, restituita dal metodo `name`, fornisce esattamente il nome della costante che rappresenta il valore dell'oggetto che esegue il metodo.

Oltre ai metodi comuni a tutti i tipi enumerativi, gli oggetti di uno specifico tipo enumerativo possono anche disporre di metodi propri. Ad esempio per il tipo enumerativo MeseDellAnno sono disponibili i seguenti metodi:

- **public MeseDellAnno successivo()**

Restituisce il valore dell'enumerativo che rappresenta il mese successivo a quello che esegue il metodo.

- **public MeseDellAnno precedente()**

Restituisce il valore dell'enumerativo che rappresenta il mese precedente a quello che esegue il metodo.

- **public int numeroGiorni()**

Restituisce il numero dei giorni del mese (28 se l'oggetto che esegue il metodo è FEBBRAIO).

- **public int numeroGiorni(int anno)**
Restituisce il numero dei giorni del mese, relativamente all'anno specificato come argomento. Pertanto, nel caso degli anni bisestili, per il mese di febbraio restituisce 29.
- **public int numeroGiorni(boolean bisestile)**
Restituisce il numero dei giorni del mese; nel caso di febbraio se l'argomento è true e restituisce 29, se è false restituisce 28.

Esercizi

4.17 Il package `prog.utili` fornisce una classe `Data` i cui oggetti rappresentano date del calendario, e i cui tipi enumerativi `MeseDellAnno` e `GiornoDellaSettimana` rappresentano rispettivamente i mesi dell'anno e i giorni della settimana. La classe `Data` mette a disposizione questi costruttori:

- **public Data()**
Costruisce una nuova data che rappresenta la data corrente.
- **public Data(int g, int m, int a)**
Costruisce una nuova data a partire da tre interi che rappresentano, nell'ordine, giorno, mese e anno.

Fra gli altri, la classe mette a disposizione questi metodi:

- **int getAnno()**
Restituisce l'intero che rappresenta l'anno della data che esegue il metodo.
- **int getGiorno()**
Restituisce l'intero che rappresenta il giorno della data che esegue il metodo.
- **int getMese()**
Restituisce l'intero che rappresenta il mese della data che esegue il metodo.
- **public GiornoDellaSettimana getGiornoDellaSettimana()**
Restituisce il valore enumerativo di tipo `GiornoDellaSettimana` che rappresenta il giorno della settimana corrispondente alla data che esegue il metodo.
- **public MeseDellAnno getMeseDellAnno()**
Restituisce il valore dell'enumerativo di tipo `MeseDellAnno` che rappresenta il mese dell'anno corrispondente alla data che esegue il metodo.
- **public boolean isInAnnoBisestile()**
Restituisce true se l'anno della data che esegue il metodo è bisestile, false altrimenti.
- **int quantoManca(Data altra)**
Restituisce il numero di giorni che intercorrono tra la data rappresentata dall'oggetto

che esegue il metodo e quella rappresentata dall'oggetto fornito tramite il parametro. In particolare, se la data rappresentata dall'oggetto che esegue il metodo precede quella fornita tramite il parametro, il risultato sarà un numero positivo (ad esempio, se le due date sono, rispettivamente, 31.12.2002 e 1.1.2003, il risultato sarà 1), se invece la segue, il risultato sarà un numero negativo (se le due date sono, rispettivamente, 1.1.2003 e 31.12.2002, il risultato sarà -1).

- `String toString()`

Restituisce la stringa che rappresenta la data corrente nel formato "gg.mm.aaaa".

Scrivete un'applicazione che legga una data di nascita e visualizzi in quale giorno della settimana cadeva e quanti giorni sono trascorsi da allora. Ad esempio, l'applicazione potrebbe dar luogo alla seguente esecuzione:

```
Inserisci la tua data di nascita:  
Giorno? 30  
Mese? 10  
Anno? 1965  
Data odierna: Giovedì 17 Febbraio 2005  
Data di nascita: Sabato 30 Ottobre 1965  
Differenza rispetto a oggi: 14355 giorni
```

- 4.18 Come si comporta l'applicazione che avete sviluppato per l'esercizio precedente se viene inserita una data futura? Modificate l'applicazione in modo che tenga conto di questa possibilità.

4.8 L'istruzione switch

Torniamo a considerare l'applicazione OccorrenzeVocali sviluppata nel Paragrafo 4.6. In quest'applicazione abbiamo utilizzato cinque istruzioni `if` innestate per selezionare, in base al valore di un'unica variabile, un'istruzione tra diverse possibili. In casi come questo risulta utile l'istruzione `switch`, che permette di selezionare l'esecuzione di un'istruzione, tra più possibili, in base al valore di un'espressione detta *selettore*.

La sintassi dell'istruzione `switch` è data da:

```
switch (espressione)
  bloccoSwitch
```

dove *bloccoSwitch* è un blocco di istruzioni, racchiuso tra parentesi graffe, in cui le istruzioni possono essere precedute da una o più etichette della forma

```
case espressioneCostante :
```

oppure:

`default:`

Affinché l'istruzione sia costruita correttamente, devono essere soddisfatti i seguenti requisiti.

- L'espressione tra parentesi dopo la parola `switch`, detta *selettore*, dev'essere di uno dei tipi `char`, `byte`, `short` o `int` oppure di un tipo enumerativo.
- Le espressioni utilizzate nelle etichette dopo la parola `case` devono essere costanti (cioè il loro valore dev'essere determinabile in fase di compilazione) assegnabili al tipo del selettore (ad esempio, se il selettore è di tipo `short`, le costanti che compaiono nelle etichette devono essere valori interi compresi fra -2^{15} e $2^{15} - 1$; se il selettore è un tipo enumerativo, le costanti che compaiono nelle etichette devono essere fra quelle che rappresentano i valori del tipo enumerativo).
- Non possono esserci due etichette in cui si utilizzano costanti con lo stesso valore.
- Non può esserci più di un'etichetta `default`.

Se una di queste condizioni viene violata, si avrà un errore *in fase di compilazione*.

L'esecuzione dell'istruzione `switch` avviene selezionando un punto d'inizio all'interno del *bloccoSwitch*, secondo le seguenti regole.

- Viene calcolato prima di tutto il valore dell'espressione scritta tra parentesi dopo la parola `switch`.
- Se c'è un'etichetta con associata una costante con valore uguale al risultato dell'espressione, si inizia a eseguire il *bloccoSwitch* a partire da questa etichetta. Se tale etichetta non c'è, ma è presente un'etichetta `default`, si inizia a eseguire il blocco di codice a partire da questa.
- Se non c'è un'etichetta con associata una costante con valore uguale a quello calcolato e nemmeno un'etichetta `default`, si passa a eseguire il codice che si trova dopo il blocco `switch`.
- Se durante l'esecuzione del *bloccoSwitch*, a partire dal punto selezionato in base alle etichette, si incontra un'istruzione `break`, si passa a eseguire il codice che si trova dopo il blocco `switch`.

Si consideri il seguente esempio:

```
switch (x) {
    case 0:
        out.println("zero");
    case 1:
        out.println("uno");
```

```
case 2:  
    out.println("due");  
    break;  
default:  
    out.println("nulla");  
}
```

dove la variabile `x` è di tipo `int`. Se `x` contiene il valore 1, l'esecuzione dell'istruzione produrrà come output:

```
uno  
due
```

Se invece `x` contiene valore 4, l'output sarà:

```
nulla
```

Ecco come può essere riscritta l'applicazione `OccorrenzeVocali` per mezzo di un'istruzione `switch`:

```
import prog.io.*;  
  
class OccorrenzeVocali {  
  
    public static void main(String [] args) {  
        //predisposizione dei canali di comunicazione  
        ConsoleInputManager in = new ConsoleInputManager();  
        ConsoleOutputManager out = new ConsoleOutputManager();  
  
        String s = in.readLine("Inserire la stringa da esaminare ");  
        int na = 0, ne = 0, ni = 0, no = 0, nu = 0;  
  
        for (int i = 0; i < s.length(); i = i + 1)  
            switch (s.charAt(i)) {  
                case 'a':  
                case 'A':  
                    na++;  
                    break;  
                case 'e':  
                case 'E':  
                    ne++;  
                    break;  
                case 'i':  
                case 'I':  
                    ni++;  
                    break;  
                case 'o':  
                case 'O':  
                    no++;  
                    break;  
                case 'u':  
                case 'U':  
                    nu++;  
                    break;  
            }  
    }  
}
```

```

        break;
    case 'o':
    case 'O':
        no++;
        break;
    case 'u':
    case 'U':
        nu++;
        break;
    }

out.println("Numero di occorrenze della vocale a: " + na);
out.println("Numero di occorrenze della vocale e: " + ne);
out.println("Numero di occorrenze della vocale i: " + ni);
out.println("Numero di occorrenze della vocale o: " + no);
out.println("Numero di occorrenze della vocale u: " + nu);
}
}

```

Si noti l'uso dell'istruzione `break` alla fine di ciascun caso. In pratica questo avviene nella maggior parte delle situazioni. Un errore banale, che però risulta difficile individuare, è proprio la dimenticanza di un `break` alla fine di un blocco di istruzioni. Tuttavia in talune situazioni può essere utile scrivere istruzioni `switch` in cui i blocchi non siano terminati da `break`.

A titolo d'esempio riportiamo un'applicazione che calcola e comunica all'utente il numero di giorni che mancano alla fine del mese e alla fine dell'anno a partire dalla data odierna; questa applicazione fornisce anche un esempio d'utilizzo dei tipi enumerativi come selettori di un'istruzione `switch`.

Per prima cosa l'applicazione costruisce la data odierna e determina il `MeseDellAnno` della data odierna. Quindi, mediante il metodo `numeroGiorni(int anno)` della classe `MeseDellAnno` determina il numero dei giorni del mese, tenendo conto dell'eventualità che l'anno sia bisestile.

Nella seconda parte dell'applicazione viene calcolato il numero di giorni che mancano alla fine dell'anno. A tale scopo, al numero dei giorni mancanti alla fine del mese considerato, vengono aggiunti via via i giorni dei mesi successivi. Anche in questo caso è utilizzata un'istruzione `switch`, il cui scopo è quello di selezionare il primo mese di cui tenere conto (quello successivo a quello della data odierna). In questo caso non viene utilizzata l'istruzione `break`: questo permette di eseguire tutti gli assegnamenti successivi; è inoltre essenziale l'ordine in cui sono elencati i diversi casi.

```

import prog.io.*;
import prog.utili.MeseDellAnno;
import prog.utili.Data;

```

```
class GiorniMancanti {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleOutputManager out = new ConsoleOutputManager();

        //costruisco la data corrispondente a oggi
        Data oggi = new Data();
        //determino il mese
        MeseDellAnno mese = oggi.getMeseDellAnno();
        //calcolo il numero dei giorni del mese
        int giorniNelMese = mese.numeroGiorni(oggi.getAnno());
        //calcolo giorni mancanti alla fine del mese
        int giorniMancanti = giorniNelMese - oggi.getGiorno();

        //calcolo giorni mancanti alla fine dell'anno
        int giorniMancantiAnno = giorniMancanti;
            //se il mese è dicembre ho finito,
            // altrimenti devo aggiungere i giorni
            //dei mesi successivi
        if (mese != MeseDellAnno.DICEMBRE){
            switch (mese.successivo()) {
                case GENNAIO: //questo caso non verrà mai eseguito
                    giorniMancantiAnno += 31;
                case FEBBRAIO:
                    giorniMancantiAnno += oggi.isInAnnoBisestile() ? 29 : 28;
                case MARZO:
                    giorniMancantiAnno += 31;
                case APRILE:
                    giorniMancantiAnno += 30;
                case MAGGIO:
                    giorniMancantiAnno += 31;
                case GIUGNO:
                    giorniMancantiAnno += 30;
                case LUGLIO:
                    giorniMancantiAnno += 31;
                case AGOSTO:
                    giorniMancantiAnno += 31;
                case SETTEMBRE:
                    giorniMancantiAnno += 30;
                case OTTOBRE:
                    giorniMancantiAnno += 31;
```

```

        case NOVEMBRE:
            giorniMancantiAnno += 30;
        case DICEMBRE:
            giorniMancantiAnno += 31;
    }
}

//comunicazione dei risultati
out.print("Mancano " + giorniMancanti + " giorni alla fine del mese");
out.println(" e " + giorniMancantiAnno + " alla fine dell'anno");
}
}

```

Si osservi la variabile `giorniMancantiAnno`: essa viene inizializzata con il numero dei giorni mancanti alla fine del mese corrente. Se il mese non è dicembre (nel qual caso non vi sono altri mesi da considerare), viene selezionato il mese successivo a quello corrente. Ad esempio, se `mese` è `SETTEMBRE`, si seleziona il caso corrispondente al valore `SETTEMBRE` ma si eseguono anche quelli corrispondenti alle etichette `OTTOBRE`, `NOVEMBRE`, `DICEMBRE`. In questo modo si eseguono, nell'ordine, i tre incrementi:

```

giorniMancantiAnno += 31;
giorniMancantiAnno += 30;
giorniMancantiAnno += 31;

```

In relazione all'utilizzo di tipi enumerativi come selettori dello `switch` osserviamo che le costanti nelle etichette sono utilizzate lasciando sottinteso il tipo enumerativo in cui sono definite. Ad esempio la seconda etichetta è `FEBBRAIO` e non `MeseDellAnno.FEBBRAIO` come ci si potrebbe aspettare. Questa è una sorta di utile abbreviazione fornita ed imposta dal compilatore. Il compilatore deduce dal tipo del selettore che si tratta di un tipo enumerativo e automaticamente si aspetta come etichette costanti di quel tipo enumerativo. Utilizzare esplicitamente il tipo enumerativo (cioè indicare ad esempio come etichetta `MeseDellAnno.FEBBRAIO`, invece di `FEBBRAIO`) dà luogo a un errore in fase di compilazione.

Esercizi

- 4.19 Sia `c` una variabile di tipo `char`. Indicate per ogni lettera maiuscola quale output è prodotto dal seguente frammento di codice, nel caso `c` contenga tale lettera:

```

switch (c) {
    case 'A':
    case 'C':
        out.println("AAA");
    case 'D':

```

4.20

4.9

Fino
getti
cui i
richie
utiliz
che,
realiz
a disp
Ha se
caso i
dalla
bili in
static
parola
date p

```

        case 'W':
            out.println("EEE");
        default:
            out.println("ddd");
        case 'H':
            break;
        case 'J':
            out.println("kkk");
            break;
        case 'G':
        case 'P':
            out.println("e");
    }
}

```

- 4.20 Costruite un'applicazione per la simulazione di una semplice calcolatrice tascabile. Il programma deve scrivere il risultato dopo avere ricevuto due numeri interi e il simbolo dell'operazione desiderata.
- 4.21 Scrivete un'applicazione che legga una stringa di caratteri e scriva il numero di occorrenze di ciascuna lettera in tale stringa. Ad esempio, leggendo la stringa `aiuola` il programma dovrà comunicare di avere incontrato due `a`, una `i`, una `l`, una `o` e una `u`.

4.9 I metodi statici

Finora abbiamo sempre descritto i metodi come implementazioni del comportamento degli oggetti e abbiamo visto che, per invocare un metodo, è necessario avere a disposizione un oggetto cui inviare la richiesta di eseguire il metodo. Esistono però casi in cui sarebbe comodo poter richiedere servizi senza disporre di oggetti ai quali chiederli. Supponiamo, ad esempio, di voler utilizzare in un programma la stringa che rappresenta un intero, cioè un valore di tipo `int`. Dato che, come abbiamo visto, i tipi primitivi non sono oggetti, non potremo chiedere a un `int` di realizzare tale compito. Tuttavia, per convertire un `int` in una stringa, la classe `String` mette a disposizione un metodo, denominato `valueOf`. A chi chiediamo di svolgere questo compito? Ha senso creare un *inutile* oggetto di tipo `String` a cui chiedere di eseguire il metodo? In questo caso il compito da eseguire non è legato a un singolo oggetto, ma è piuttosto un servizio fornito dalla classe `String` stessa. In situazioni come questa, Java consente di definire metodi utilizzabili indipendentemente dall'esistenza di un oggetto. Questi metodi prendono il nome di *metodi statici* o *metodi di classe*, e sono riconoscibili per il fatto che nella loro intestazione compare la parola riservata `static`. Per il resto l'intestazione di tali metodi segue le medesime convenzioni date per gli altri metodi.

Ad esempio il metodo `valueOf` della classe `String` è così documentato:

- `public static String valueOf(int i)`

Restituisce il riferimento alla stringa che rappresenta il valore dell' `int` fornito come argomento.

Diversamente dagli altri metodi, i metodi statici sono invocati utilizzando come destinatario del messaggio il nome della classe anziché il riferimento a un oggetto. Quindi l'invocazione di un metodo statico ha la seguente sintassi:

nome_classe.nome_methodo(lista_argomenti)

Ad esempio, per ottenere la stringa che rappresenta il valore contenuto in una variabile `x` di tipo `int`, possiamo usare quest'istruzione:

```
String rappresentazione = String.valueOf(x);
```

È possibile invocare i metodi statici anche utilizzando come destinatario un oggetto della classe cui appartiene il metodo statico; questa scelta è però sconsigliabile in quanto rende più difficile la lettura dei programmi. Il fatto che l'invocazione di un metodo abbia come destinatario una classe anziché un oggetto evidenzia immediatamente che si tratta di un metodo statico.

Molte classi mettono a disposizione metodi statici. Uno degli impieghi più comuni di tali metodi è quello di fornire un modo per costruire oggetti della classe stessa a partire da oggetti o valori di altro tipo, come nel caso del metodo `valueOf` che abbiamo considerato qui. In altre circostanze i metodi statici forniscono invece operazioni utili su oggetti o su tipi primitivi. Ad esempio la classe `Math` del package `java.lang` mette a disposizione metodi per realizzare le operazioni numeriche di base come l'esponenziale, il logaritmo, la radice quadrata e le funzioni trigonometriche, che operano su tipi primitivi. Altri esempi di metodi statici utili sono quelli definiti nelle cosiddette classi involucro, che vedremo più avanti.

Infine, i metodi statici vengono frequentemente utilizzati per definire proprietà che influenzano il comportamento di tutti gli oggetti di una classe. Consideriamo ad esempio il metodo `toString` della classe `Data` descritta nell'Esercizio 4.17. Tale metodo restituisce la stringa che rappresenta la data che esegue il metodo. La stringa codifica di default la data come "gg.mm.aaaa", cioè nel formato GMA (Giorno Mese Anno) utilizzando il punto come separatore. Il formato di rappresentazione di una data deve essere uniforme all'interno di un'applicazione: volendo cambiare il formato di rappresentazione o il carattere di separazione vorremmo poter modificare in una sola volta il comportamento di tutti gli oggetti di tipo `Data` utilizzati dall'applicazione. Questo è ciò che consentono di fare i metodi statici `setFormato` e `setSeparatore` della classe `Data`. Sono anche definiti i relativi metodi `getFormato` e `getSeparatore` per ottenere informazioni sul formato e sul separatore attualmente utilizzati. I formati possibili sono rappresentati dalle costanti del tipo enumerativo `FormatoData` definito nel package `prog.utili`.

- `public static void setFormato(FormatoData f)`

Definisce il formato utilizzato per la costruzione della stringa che rappresenta la data. I

valori possibili dell'argomento sono le costanti del tipo enumerativo `FormatoData` che sono AMG (Anno Mese Giorno), GMA (Giorno Mese Anno), MGA (Mese Giorno Anno) ed ESTESO.

- `public static FormatoData getFormato()`
Restituisce la costante enumerativa di `FormatoData` che descrive il formato utilizzato per la costruzione della stringa che rappresenta la data.
- `public static void setSeparatore(char c)`
Definisce il separatore utilizzato per separare giorno, mese e anno nella stringa che rappresenta la data. Di default il separatore è il punto.
- `public static char getSeparatore()`
Restituisce il separatore utilizzato per separare giorno mese e anno nella stringa che rappresenta la data.

Ad esempio, la seguente applicazione stampa la data corrente nei tre formati disponibili utilizzando come separatore il carattere /:

```
import prog.io.*;
import prog.utili.Data;
import prog.utili.FormatоГData;

class FormatоГData {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        Data.setSeparatore('/');
        Data d = new Data();
        out.println("Formato: " + Data.getFormato() +
                   ", data: " + d.toString());
        Data.setFormato(FormatоГData.MGA);
        out.println("Formato: " + Data.getFormato() +
                   ", data: " + d.toString());
        Data.setFormato(FormatоГData.AMG);
        out.println("Formato: " + Data.getFormato() +
                   ", data: " + d.toString());
        Data.setFormato(FormatоГData.ESTESO);
        out.println("Formato: " + Data.getFormato() +
                   ", data: " + d.toString());
    }
}
```

Esempio: somma di una sequenza di numeri forniti tramite una stringa

Utilizzeremo ora il metodo statico `parseInt`, della classe `Integer`, per convertire stringhe che rappresentano numeri interi in `int`. Il problema di convertire valori numerici che vengono forniti tramite stringhe si verifica spesso nella scrittura dei programmi. Supponiamo di voler sommare una sequenza di numeri interi che vengono forniti al programma in una stringa, separati da spazi. Ad esempio supponiamo che sequenza sia un riferimento alla stringa "10 23 2 11 28". Per effettuare la somma dovremo estrarre dalla stringa, una alla volta, le sottostringhe che rappresentano i numeri, convertire in `int` tali sottostringhe e utilizzare gli interi così ottenuti per calcolare la somma.

Per estrarre le sottostringhe possiamo utilizzare la classe `StringTokenizer` di Java. Si tratta di una classe contenuta nel package `java.util`, che fornisce varie classi di utilità. `StringTokenizer` mette a disposizione, fra gli altri, il seguente costruttore:

- `StringTokenizer(String str, String delim)`

Costruisce un estrattore di token per la stringa `str`, che utilizza i caratteri specificati nella stringa `delim` come separatori.

Sostanzialmente uno `StringTokenizer` è un oggetto che fornisce un meccanismo per estrarre sottostringhe, dette *token*, dalla stringa fornita come argomento. I token sono determinati a partire dai delimitatori specificati nella costruzione dello `StringTokenizer`. Nel nostro esempio esiste un unico delimitatore costituito dallo spazio; quindi, la dichiarazione

```
 StringTokenizer stk = new StringTokenizer(sequenza, " ");
```

associa a `stk` il riferimento all'istanza di `StringTokenizer` che ci serve. Dato che lo `StringTokenizer` `stk` ha come delimitatore lo spazio, esso interpreta la stringa `sequenza` come la sequenza di token "10", "23", "2", "11" e "28".

La classe `StringTokenizer` mette a disposizione un metodo `nextToken` che in invocazioni successive restituisce i token che compaiono nella stringa. Ad esempio la prima invocazione di `nextToken` su `stk` restituisce il token "10", la seconda il token "23", e così via. Quindi per estrarre tutti i token contenuti nella stringa potremmo utilizzare un ciclo del tipo:

```
 while (ci sono ancora token) {
    preleva il prossimo token
    elaboralo
}
```

Per esprimere la condizione del ciclo possiamo utilizzare il metodo `hasMoreTokens` della classe `StringTokenizer`, che non prevede argomenti e restituisce un `boolean`. Questo metodo restituisce `true` se non sono stati estratti tutti i token dall'istanza di `StringTokenizer` che lo esegue, e restituisce `false` in caso contrario. Quindi possiamo riscrivere il ciclo nel modo seguente:

```
 while (stk.hasMoreTokens()) {
    preleva il prossimo token
```

```
    elaboralo
}
```

Ora, per completare l'applicazione, dobbiamo estrarre un token, convertirlo in un `int` e utilizzare tale valore per calcolare la somma.

Per estrarre un token da un'istanza di `StringTokenizer` possiamo utilizzare il metodo `nextToken`. Questo metodo restituisce una stringa: dovremo quindi memorizzarne il riferimento in una variabile di tipo `String`. Per trasformare una stringa in un intero possiamo utilizzare il metodo statico `parseInt` della classe `Integer`, che riceve come argomento la stringa da convertire e restituisce l'intero che essa rappresenta. Ad esempio, fornendo come argomento la stringa "124" il metodo restituisce l'`int` 124. Qualora la stringa fornita come argomento non sia composta esclusivamente da cifre, in fase di esecuzione questo metodo dà luogo a un errore del tipo `NumberFormatException`. A questo punto possiamo scrivere il cuore della nostra applicazione:

```
 StringTokenizer stk = new StringTokenizer(sequenza, " ");
 int somma = 0;

 while (stk.hasMoreTokens()) {
     String token = stk.nextToken();
     somma = somma + Integer.parseInt(token);
 }
```

Riassumendo, la nostra applicazione è:

```
import prog.io.*;
import java.util.StringTokenizer;

class SommaNumeriDaStringa {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleOutputManager out = new ConsoleOutputManager();
        ConsoleInputManager in = new ConsoleInputManager();

        //lettura della sequenza
        String sequenza = in.readLine("Inserisci una sequenza di numeri " +
            "interi separati da spazio: ");

        //costruzione dell'estrattore di token
        StringTokenizer stk = new StringTokenizer(sequenza, " ");

        //calcolo della somma
        int somma = 0;
```

```

        while (stk.hasMoreTokens()){
            String token = stk.nextToken();
            somma = somma + Integer.parseInt(token);
        }

        //comunicazione del risultato
        out.println("La somma dei numeri nella sequenza è: " + somma);
    }
}

```

Si osservi che, mentre `Integer` appartiene al package `java.lang`, e quindi non è necessario importarla esplicitamente, la classe `StringTokenizer` appartiene al package `java.util` e va importata.

Notiamo che questo programma funziona correttamente a patto che la stringa inserita dall'utente abbia il formato previsto. In caso contrario è segnalato un errore in fase di esecuzione. Sarebbe ovviamente opportuno che un'applicazione non interrompesse la sua esecuzione per un errore nel formato dei dati in ingresso. Per controllare che la sequenza inserita dall'utente abbia il formato previsto potremmo inserire nel nostro programma un frammento di codice in modo da verificare che sia composta solo da cifre e da spazi. Osserviamo ancora che, costruendo i token, l'istanza di `StringTokenizer` ignora tutte le occorrenze di un delimitatore; se forniamo come input una sequenza in cui i numeri sono separati da più spazi, il programma funziona comunque correttamente.

Al termine di questo paragrafo riassumiamo per comodità i due metodi di `StringTokenizer` utilizzati.

- `public boolean hasMoreTokens()`
Controlla se nella stringa associata all'istanza di `StringTokenizer` che esegue il metodo esistono altri token disponibili. Se il metodo restituisce `true`, la prossima chiamata del metodo `nextToken` restituirà il prossimo token della stringa.
- `public String nextToken()`
Restituisce il prossimo token della stringa associata all'istanza di `StringTokenizer` che esegue il metodo. Se tale istanza non contiene altri token, si verifica un errore.

Esercizi

4.22 Scrivete un'applicazione che legga due numeri interi, calcoli la loro somma e visualizzi i numeri e la loro somma opportunamente incolonnati, come ad esempio:

```

932+
 84=
-----
 1016

```

- 4.23 Modificate l'applicazione `SommaNumeriDaStringa` in modo che venga richiesto all'utente di reinserire la sequenza se essa non è costituita esclusivamente da cifre e spazi. Per verificare se un carattere è una cifra decimale si può utilizzare il metodo statico `isDigit` della classe `java.lang.Character`, così descritto:

- `public static boolean isDigit(char ch)`

Determina se il carattere specificato come argomento è una cifra. Il metodo restituisce `true` se il carattere è una cifra, e `false` in caso contrario.

- 4.24 Scrivete un'applicazione `SommaFrazioneDaStringa` che, letta in ingresso una stringa in cui è specificata una sequenza di frazioni, le sommi e visualizzi la somma a video. Il formato della stringa potrebbe essere quello dell'esempio seguente:

$$3/4 + 1/2 + 7/8 + 1/4$$

Il programma dovrà chiedere di reinserire la stringa fornita in ingresso nel caso essa non abbia il formato previsto.

- 4.25 Modificate l'applicazione precedente in modo che sia possibile specificare anche numeri interi. Il programma deve cioè accettare una stringa del tipo

$$3/4 + 2 + 7 + 1/4$$

che va intesa come la somma delle frazioni

$$3/4 + 2/1 + 7/1 + 1/4$$

- 4.26 Scrivete un'applicazione che legga due importi (dei quali l'utente specifica euro e centesimi) e visualizzi la loro somma, la loro media, la differenza tra il maggiore e il minore. Utilizzate la classe `Importo` del package `prog.utili`.

- 4.27 Modificate l'applicazione precedente in modo che gli importi vengano inseriti in lire, anziché in euro. I calcoli dovranno essere invece effettuati in euro, e i risultati comunicati in euro. Per la conversione da lire a euro, la classe `Importo` fornisce il metodo statico `fromLire` che riceve come argomento un numero intero (importo in lire) e restituisce come risultato un riferimento a un nuovo oggetto di tipo `Importo` (che rappresenta l'equivalente in euro).

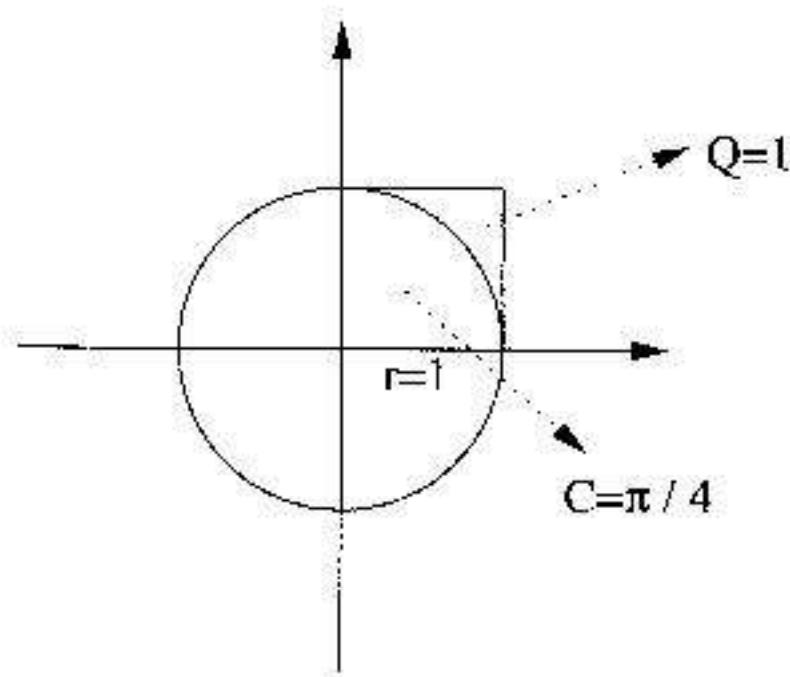
- 4.28 Costruite una classe con un metodo `main` per la soluzione delle equazioni di secondo grado. Per il calcolo della radice quadrata è disponibile nella classe `Math` un metodo statico di nome `sqrt`, che riceve un argomento di tipo `double` e restituisce un risultato di tipo `double`.

4.29 La classe `Math` del package `java.lang` fornisce un metodo statico `log` che riceve come argomento un valore `double` e ne restituisce il logaritmo naturale (sempre rappresentato come `double`). Scrivete un'applicazione che riceva in ingresso due numeri in virgola mobile x e y e calcoli (in maniera approssimata) l'integrale da x a y della funzione logaritmo. Il calcolo può basarsi sulla seguente idea (nell'ipotesi che $y \geq x \geq 1$):

- Si prenda in esame un rettangolo con base coincidente con l'intervallo considerato, cioè da x a y , e con altezza uguale al valore massimo della funzione nell'intervallo (in questo caso $\log(y)$).
- Si generano molti punti a caso all'interno di questo rettangolo.
- Il rapporto tra il numero di punti che si trovano al di sotto della funzione e il numero di punti generati sarà circa uguale al rapporto tra l'integrale da calcolare e l'area del rettangolo.

Implementate quest'idea nella vostra applicazione, utilizzando la classe `Random`, descritta nell'Esercizio 3.28.⁷ La classe dispone anche di un metodo `nextDouble`, privo di argomenti, che restituisce un valore `double` generato a caso, maggiore o uguale a zero e minore di 1.

4.30 Si consideri un cerchio di raggio 1 il cui centro coincide con l'origine di una coppia di assi cartesiani.



Scrivete un'applicazione che generando una sequenza di punti a caso di coordinate x e y comprese fra 0 e 1 calcoli la frequenza dei punti che cadono nel quarto di cerchio di circonferenza. Si ricordi che la frequenza è data dal rapporto fra i casi favorevoli (i punti che cadono nel quarto di circonferenza) e i casi totali (i punti generati).

4.31 Estendendo l'applicazione dell'esercizio precedente è possibile calcolare un'approssimazione della costante π utilizzando un metodo Monte Carlo. A tale scopo è sufficiente osservare che la probabilità P che il punto cada nel quarto di circonferenza è data dal rapporto fra l'area C del quarto di circonferenza e l'area Q del quadrato di lato 1. Quindi $P = \frac{\pi}{4}$,

⁷ Algoritmi che utilizzano generatori casuali per il calcolo, prendono anche il nome di "metodi Monte Carlo".

da cui $\pi = 4P$. Come approssimazione del valore della probabilità P si può utilizzare la frequenza calcolata nel modo descritto nell'esercizio precedente.

- 4.32 Confrontate il valore di π calcolato con il metodo dell'esercizio precedente con la costante `Math.PI`. In particolare potete costruire un'applicazione che generi via via punti di coordinate comprese fra 0 e 1. Ogni volta che viene generato un nuovo punto, l'applicazione ricalcola il valore della frequenza, considerando tutti i punti generati fino a quel momento, e, in base a esso, l'approssimazione di π . Inoltre, l'applicazione calcola la differenza in valore assoluto tra l'approssimazione di π e `Math.PI`. Al crescere del numero di punti generati questa differenza dovrebbe ridursi rapidamente. Scrivete l'applicazione in due versioni: nella prima l'applicazione termina dopo aver generato un numero di punti stabilito preliminarmente dall'utente, nella seconda l'applicazione termina quando la differenza tra l'approssimazione di π e `Math.PI` è minore di un valore inserito preliminarmente dall'utente. Per calcolare il valore assoluto, la classe `Math` fornisce una metoda statico `abs` che riceve un `double` come argomento e restituisce un `double`.

4.10 Le classi involucro

L'introduzione dei tipi primitivi nel linguaggio Java è dovuta principalmente a questioni di efficienza. Avremmo potuto rappresentare come oggetti anche i dati dei tipi primitivi. Ad esempio possiamo immaginare di avere oggetti che rappresentano i numeri interi. L'accesso a un numero intero, rappresentato direttamente in una variabile, è chiaramente più veloce dell'accesso tramite un riferimento.

In talune situazioni, tuttavia, può risultare utile rappresentare dati di tipi primitivi sotto forma di oggetti. A tale scopo, nel package della libreria standard `java.lang` esiste una classe associata a ogni tipo primitivo, che permette di rappresentare i dati di quel tipo come oggetti. Queste classi sono dette *classi involucro*. In esse sono inoltre contenuti campi e metodi utili per trattare gli oggetti del tipo, o anche i dati primitivi.

Consideriamo come esempio la classe `Integer`, del package `java.lang`, associata al tipo primitivo `int`. La classe dispone di due costruttori:

- `public Integer(int x)`
Costruisce un nuovo oggetto che rappresenta il numero intero fornito come argomento.
- `public Integer(String str)`
Costruisce un nuovo oggetto che rappresenta il numero intero fornito sotto forma di stringa tramite l'argomento. Se la stringa non rappresenta un numero intero si verifica un errore.

Tra i numerosi metodi a disposizione segnaliamo:

- `public int compareTo(Integer altro)`
Confronta l'intero rappresentato dall'oggetto che esegue il metodo con quello fornito tramite il parametro. Restituisce zero se i due valori coincidono, un valore minore di zero se il valore rappresentato dall'oggetto è minore di quello fornito tramite il parametro, un valore maggiore di zero in caso contrario.

- `public int intValue()`
Restituisce un `int` uguale al valore rappresentato dall'oggetto.
- `public long longValue()`
Restituisce un `long` uguale al valore rappresentato dall'oggetto.
- `public static int parseInt(String s)`
Restituisce un `int` uguale al valore rappresentato dalla stringa fornita come argomento.
Ad esempio, se la stringa è "1234", restituisce il valore 1234. Nel caso la stringa non rappresenti un intero, si verifica un errore in fase di esecuzione.

La classe contiene inoltre due *campi statici*, le costanti `MIN_VALUE` e `MAX_VALUE`, che corrispondono al minimo e al massimo valore intero rappresentabile tramite un `int`. Tali valori sono disponibili utilizzando espressioni con una notazione simile a quella per accedere ai metodi statici, con la seguente sintassi:

nome_classe.nome_campo

Ad esempio, l'espressione `Integer.MAX_VALUE` ha come tipo `int` e ha come valore il massimo intero rappresentabile in Java tramite un `int`.

Il linguaggio Java fornisce dei meccanismi automatici di conversione implicita fra tipi primitivi e corrispondenti classi involucro che prendono il nome di *autoboxing* e *unboxing*.⁸

Consideriamo, ad esempio, le seguenti istruzioni:

```
Integer i = new Integer(123);
int j = i.intValue();
```

Per prima cosa abbiamo utilizzato il costruttore della classe `Integer` che riceve un argomento di tipo `int`, per costruire l'oggetto che rappresenta l'intero 123; quindi abbiamo invocato il metodo `intValue()` dell'oggetto costruito, per ottenere, sotto forma di `int`, il valore in esso contenuto. Grazie al meccanismo dell'unboxing è possibile riscrivere la seconda istruzione in modo più compatto come:

```
int j = i;
```

In questo caso è il compilatore che provvede ad effettuare le operazioni necessarie a "togliere dalla scatola", rappresentata dall'oggetto di tipo `Integer`, il corrispondente valore di tipo `int`.

Il meccanismo dell'unboxing del tipo `Integer` viene applicato automaticamente dal compilatore quando viene utilizzato un riferimento a un oggetto di tipo `Integer` in un punto in cui è ammessa un'espressione di tipo `int`. Nel caso del codice precedente l'oggetto di tipo `Integer` compare alla destra di un assegnamento a una variabile di tipo `int`; dato che il compilatore in quella posizione si aspetta un'espressione di tipo `int` provvede automaticamente a effettuare la conversione. Un altro esempio di unboxing è fornito dalla seguente istruzione:

```
double d = 2.0 + new Integer("123");
```

⁸ Questi meccanismi sono stati introdotti a partire dalla versione 5 del linguaggio.

Anche in questo caso il compilatore effettua un unboxing in quanto un oggetto di tipo `Integer` compare in un punto del codice in cui il compilatore può ammettere un valore di tipo `int` (che viene promosso a `double` prima che venga effettuata la valutazione dell'espressione).

L'*autoboxing* è invece il meccanismo per cui un valore di tipo primitivo viene automaticamente "inscatolato" in un oggetto della classe involucro corrispondente quando compare in un punto del codice in cui il compilatore si aspetta un riferimento del tipo della classe involucro. Ad esempio, nell'istruzione

```
Integer i = 123;
```

il valore di tipo `int` alla destra dell'assegnamento viene posto automaticamente in un nuovo oggetto di tipo `Integer`; quindi, a tutti gli effetti, l'istruzione precedente è equivalente a:

```
Integer i = new Integer(123);
```

Per concludere questa sezione vogliamo osservare che, se i meccanismi di unboxing e autoboxing rendono più evanescente la distinzione fra tipi primitivi e tipi riferimento, comunque non la eliminano (non è comunque possibile invocare un metodo su un'espressione di tipo primitivo). Inoltre per questioni di efficienza unboxing e autoboxing andrebbero utilizzati solo in situazioni in cui sia indispensabile passare dalla rappresentazione tramite il tipo primitivo alla rappresentazione tramite oggetti del corrispondente tipo involucro o viceversa. Vedremo nei prossimi capitoli alcuni esempi di utilizzo appropriato di questi meccanismi.

Esempio: conteggio del numero di lettere minuscole in una stringa

Come esempio di utilizzo di metodi statici offerti dalle classi involucro presentiamo un'applicazione che, letta una stringa di caratteri da tastiera, calcola e comunica all'utente quanti fra i caratteri che la costituiscono sono lettere minuscole. Ricordiamo che i caratteri di una stringa sono numerati a partire dalla posizione 0. Se il riferimento alla stringa letta si trova in una variabile `s` dichiarata di tipo `String`, è sufficiente far scorrere i caratteri della stringa, dalla posizione 0 alla posizione `s.length() - 1`, e contare quanti di questi sono lettere minuscole. Utilizzeremo dunque un ciclo della forma:

```
for (int i = 0; i < s.length(); i = i + 1)
    if (la posizione i di s contiene una lettera minuscola)
        incrementa il contatore delle minuscole;
```

Per ottenere un singolo carattere di una stringa, possiamo chiedere alla stringa stessa di fornircelo usando il seguente metodo della classe `String`:

- `public char charAt (int indice)`

Restituisce il carattere nella posizione specificata della stringa che esegue il metodo. Gli indici ammissibili sono compresi fra 0 e `length() - 1`. Il primo carattere della stringa ha indice 0, il secondo ha indice 1, e così via. Se `indice` ha un valore negativo o maggiore o uguale a `length`, il metodo dà luogo a un errore in fase di esecuzione.

Utilizzando questo metodo possiamo riscrivere il ciclo come:

```
for (int i = 0; i < s.length(); i = i + 1) {
    char c = s.charAt(i);
    if (c contiene una lettera minuscola)
        incrementa il contatore delle minuscole;
}
```

Allo scopo di realizzare la condizione dell'if possiamo utilizzare la classe involucro `Character` associata al tipo `char`, definita nel package `java.lang`. Per verificare le proprietà di un carattere, questa classe mette a disposizione diversi metodi statici. Ad esempio fornisce un metodo statico di nome `isLowerCase` che permette di verificare se un carattere è una lettera minuscola o no. Tale metodo riceve come argomento un carattere e restituisce un `boolean`, in particolare restituisce `true` se e solo se il carattere fornito come argomento è una lettera minuscola. Poiché è un metodo statico, la sua invocazione avviene scrivendo `Character.isLowerCase` seguito dall'argomento.

Ecco il testo completo di una classe `ContaMinuscole` contenente un metodo `main` sviluppato secondo quanto indicato sopra:

```
import prog.io.*;

class ContaMinuscole {

    public static void main(String [] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //lettura della stringa
        String s = in.readLine("Inserire la stringa da esaminare ");

        //conteggio delle lettere minuscole
        int nMinuscole = 0;
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (Character.isLowerCase(c))
                nMinuscole = nMinuscole + 1;
        }

        //comunicazione del risultato
        out.print("La stringa " + s + " contiene ");
        out.println(nMinuscole + " lettere minuscole");
    }
}
```

Per concludere questo paragrafo proponiamo la descrizione di alcuni metodi della classe `Character`.

- `public static boolean isDigit(char ch)`
Determina se il carattere specificato come argomento è una cifra. Il metodo restituisce `true` se il carattere è una cifra, e `false` in caso contrario.
- `public static boolean isLetter(char ch)`
Determina se il carattere specificato come argomento è una lettera. Il metodo restituisce `true` se il carattere è una lettera, e `false` in caso contrario.
- `public static boolean isLowerCase(char ch)`
Determina se il carattere specificato come argomento è una lettera minuscola. Il metodo restituisce `true` se il carattere è una lettera minuscola, e `false` in caso contrario.
- `public static boolean isLetterOrDigit(char ch)`
Determina se il carattere specificato come argomento è una lettera o una cifra. Il metodo restituisce `true` se il carattere è una lettera o una cifra, e `false` in caso contrario.
- `public static boolean isUpperCase(char ch)`
Determina se il carattere specificato come argomento è una lettera maiuscola. Il metodo restituisce `true` se il carattere è una lettera maiuscola, e `false` in caso contrario.
- `public static char toLowerCase(char ch)`
Converte il carattere specificato come argomento nel corrispondente carattere minuscolo.
- `public static char toUpperCase(char ch)`
Converte il carattere specificato come argomento nel corrispondente carattere maiuscolo.

Esercizi

4.33 Sia `c` una variabile di tipo `char`. Senza usare l'operatore `!` di negazione, ma utilizzando esclusivamente gli operatori `&&` e `||` e gli operatori di confronto, esprimete le seguenti condizioni:

- (1) `c` contiene una lettera maiuscola
- (2) `c` contiene una lettera minuscola
- (3) `c` non contiene una lettera maiuscola
- (4) `c` non contiene una lettera minuscola
- (5) `c` contiene una lettera
- (6) `c` non contiene una lettera.

4.34 Esprimete le condizioni elencate nell'esercizio precedente ricorrendo ai metodi offerti dalla classe involucro `Character` (per conoscere i metodi disponibili consultate la documentazione delle librerie).

4.35 Studiate e interpretate la differenza fra le seguenti espressioni di tipo `boolean`:

```
123 == 123  
123 == new Integer(123)  
new Integer(123) = new Integer(123)
```

4.36 Scrivete un'applicazione che legga una sequenza di numeri interi e, al termine, visualizzi il numero più grande e il numero più piccolo presenti nella sequenza. Svolgete l'esercizio in due modi:

- rappresentando gli interi mediante variabili del tipo primitivo `int`,
- rappresentando gli interi come oggetti della classe `Integer` (package `java.lang`) ed utilizzando, per i confronti, i metodi `compareTo` e `equals` forniti da tale classe.

Supponete che l'inserimento dello zero indichi la fine della sequenza (pertanto zero non deve essere considerato come elemento della sequenza).

Array e collezioni

In ciascuna delle applicazioni che abbiamo scritto nei capitoli precedenti la quantità di dati memorizzati era nota a priori, anche quando non era nota la quantità di dati da elaborare. Si pensi ad esempio al calcolo della somma di una sequenza di frazioni, presentato nel Paragrafo 3.5: per svolgere il proprio compito, l'applicazione non ha bisogno di memorizzare la sequenza man mano letta, ma solamente la sua somma. In questo modo, è sufficiente disporre di una variabile di tipo `Frazione` per la somma, oltre che di un'altra variabile per l'ultima frazione letta.

Non sempre il compito richiesto permette, come in questo esempio, di “dimenticarsi” della sequenza. Se, ad esempio, venisse richiesto di visualizzare la sequenza letta, sarebbe indispensabile memorizzarla durante la lettura.

Molte delle applicazioni che si scrivono nella pratica hanno la necessità di memorizzare *collezioni* di dati. In base alle operazioni che su questi dati devono essere applicate, queste collezioni vengono organizzate in *strutture dati* appropriate. Java offre diverse strutture, alcune, come gli array, implementate direttamente nel linguaggio, altre nelle librerie standard.

Nella prima parte di questo capitolo presentiamo gli array, un tipico meccanismo per strutturare i dati, offerto dalla maggior parte dei linguaggi di programmazione. Nella seconda parte introduciamo invece alcuni semplici esempi di collezioni di oggetti. Lo strumento principale grazie al quale possiamo trattare le collezioni in questo punto del testo sono i *tipi generici*, introdotti a partire dalla versione 5 del linguaggio. Essi permettono un utilizzo estremamente semplice delle collezioni anche da parte di chi si avvicini per la prima volta a un linguaggio di programmazione, evitando di concentrare l'attenzione su numerosi dettagli, come richiesto con gli strumenti disponibili nelle versioni precedenti del linguaggio.

5.1 Array di oggetti

Un *array* è un insieme ordinato di variabili dello *stesso tipo* (detto *tipo base*), ognuna delle quali è accessibile specificando la posizione in cui si trova.¹ Il tipo base di un array può essere sia un tipo primitivo sia un tipo riferimento, nel qual caso parliamo di *array di oggetti*.

¹ Spesso il termine *array* è tradotto in italiano con *vettore*. Dato che in Java tale termine viene utilizzato per gli oggetti della classe `Vector`, per evitare confusioni utilizzeremo il termine *array* anche in italiano.

Una variabile in grado di contenere un riferimento a un array può essere dichiarata scrivendo il nome del tipo base seguito da parentesi quadre aperte e chiuse, seguito dal nome della variabile. Ad esempio:

```
String[] nomi;
Frazione[] frazioni;
```

sono due dichiarazioni: la prima dichiara la variabile `nomi` come una variabile riferimento a un array di oggetti `String`, la seconda dichiara `frazioni` come una variabile riferimento a un array di oggetti di tipo `Frazione`.

In Java gli array sono oggetti, e quindi vanno creati utilizzando l'operatore `new`. La sintassi dell'operazione di creazione è però diversa da quella prevista per gli altri oggetti. Per creare un oggetto array dobbiamo utilizzare l'operatore `new` seguito dal nome del tipo base e da un'espressione di tipo `int`, tra parentesi quadre, il cui risultato è il numero di elementi dell'array. Ad esempio, l'espressione:

```
new Frazione[4]
```

restituisce un riferimento a un array di quattro posizioni preposte a contenere riferimenti a oggetti di tipo `Frazione`. Tale riferimento può essere assegnato a una variabile opportunamente dichiarata. Ad esempio, possiamo scrivere:

```
frazioni = new Frazione[4];
```

In questo caso viene costruito un array di 4 posizioni e il riferimento a questo array è assegnato alla variabile `frazioni`. Ogni oggetto di tipo array ha memorizzata nel suo stato l'informazione relativa alla dimensione, cioè al numero di elementi, dell'array. Tale informazione si trova in un *campo* (cioè, come vedremo, in una variabile interna all'oggetto) di nome `length` e di tipo `int`. Ad esempio:

```
frazioni.length
```

vale 4. Si osservi che `length` è un campo dell'oggetto array (e non un metodo come nel caso della classe `String`), e quindi non dev'essere seguito dalle parentesi tonde.

Ciascun elemento dell'array è accessibile utilizzando un *selettore* per specificare la posizione dell'elemento. L'elemento viene selezionato scrivendo il riferimento all'array seguito da un'espressione di tipo `int`, tra parentesi quadre, il cui risultato è il selettore.² Le posizioni di un array sono contate a partire da zero. Dunque `frazioni[0]` indica il *primo* elemento dell'array cui si riferisce `frazioni`, mentre `frazioni[3]` indica il *quarto e ultimo* elemento dell'array.

In generale, se `x` è una variabile di tipo `int`, `frazioni[x]` indica l'elemento dell'array cui si riferisce `frazioni`, la cui posizione è uguale al valore di `x` (contando da zero). Se `x` non corrisponde a nessuna posizione dell'array (in questo caso, se `x` è negativo o maggiore o uguale a 4), si verifica un errore in esecuzione.

² Il tipo del selettore dev'essere un'espressione di tipo `int`. Possono essere utilizzate anche espressioni di tipo `short`, `byte`, o `char` poiché, utilizzate come indici di un array, vengono promosse automaticamente a `int`. Al contrario, il compilatore non permette di utilizzare espressioni di tipo `long` come selettore.

Ad esempio gli assegnamenti:

```
frazioni[0] = new Frazione(1,4);
frazioni[1] = new Frazione(2,4);
frazioni[2] = new Frazione(3,4);
frazioni[3] = new Frazione(4,4);
```

hanno l'effetto di assegnare la frazione $\frac{1}{4}$ al primo elemento dell'array, la frazione $\frac{2}{4}$ al secondo, e così via. Dopo questi assegnamenti la situazione in memoria può quindi essere rappresentata come nella Figura 5.1.

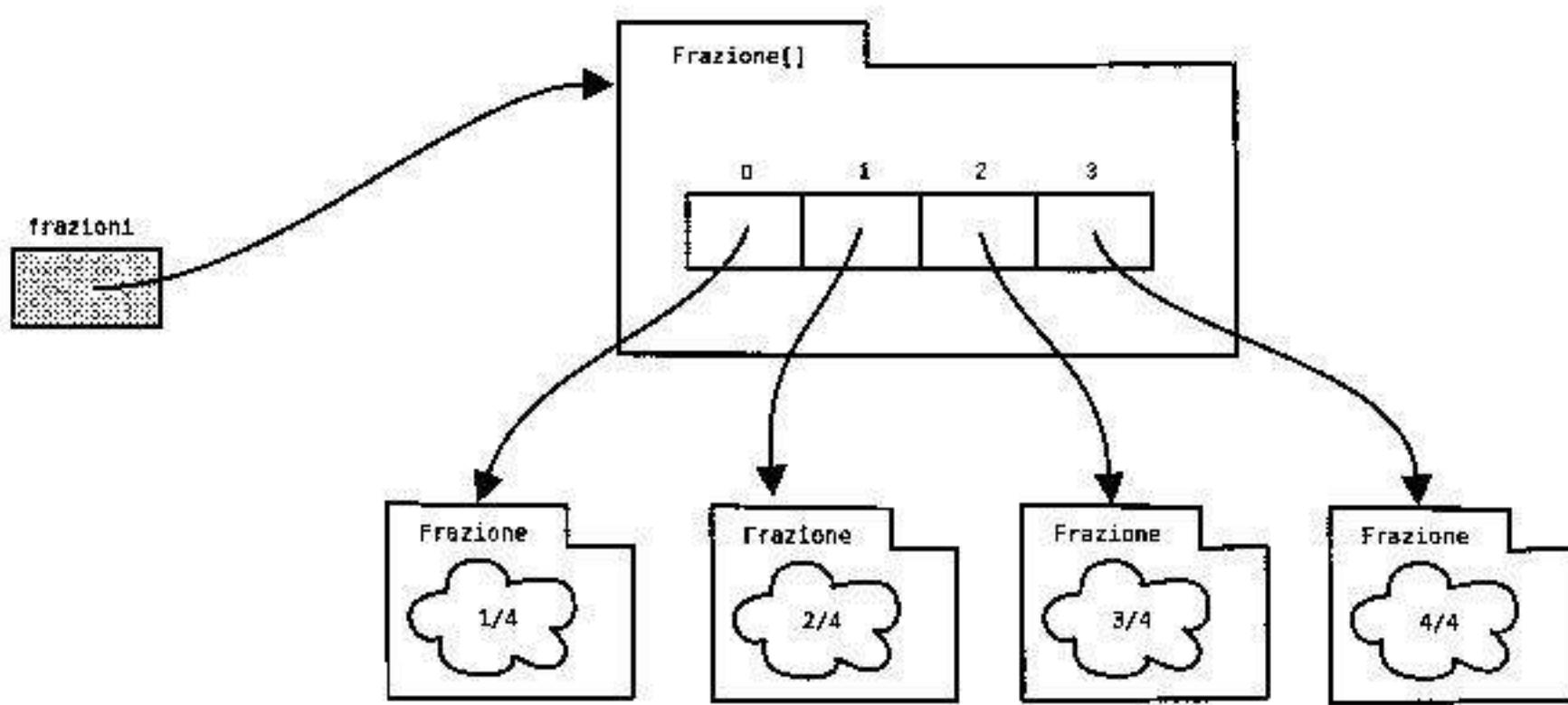


Figura 5.1 Array di frazioni.

Il tentativo di accedere a una componente non definita dell'array, ad esempio tramite l'assegnamento

```
frazioni[4] = new Frazione(5,4);
```

causa un errore in fase di esecuzione. Si osservi che la precedente istruzione verrebbe comunque compilata correttamente; il compilatore non esegue alcun controllo sui possibili valori utilizzati come indici dell'array. Solo in fase di esecuzione, cercando di accedere alla quinta componente dell'array `frazioni` (specificata appunto da `frazioni[4]`), la Java Virtual Machine si accorge che questa corrisponde a una posizione assente, e quindi segnala un errore.

È possibile inizializzare un array direttamente in fase di dichiarazione specificando fra parentesi graffe la sequenza di valori che costituiscono gli elementi dell'array. Ad esempio, possiamo ricreare la situazione rappresentata nella figura tramite la seguente definizione (dichiarazione più inizializzazione) della variabile `frazioni`:

```
Frazione[] frazioni = {new Frazione(1,4), new Frazione(2,4),
                      new Frazione(3,4), new Frazione(4,4)};
```

Si osservi che in questo caso la dimensione dell'array, cioè il numero di elementi che lo compongono, viene dedotta dal compilatore a partire dal numero di elementi specificati nella sequenza.

Esempio: lettura di una sequenza di frazioni

Vediamo com'è possibile operare sugli array scrivendo una semplice applicazione che legge da tastiera una sequenza di frazioni e la riscrive a video:

```
import prog.io.*;
import prog.utili.Frazione;

class SequenzaFrazioni {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        final int MAX = 10;
        Frazione[] frazioni = new Frazione[MAX];

        //fase lettura
        for (int pos = 0; pos < MAX; pos++) {
            out.println("Lettura della frazione " + (pos + 1));
            int num = in.readInt("Numeratore: ");
            int den = in.readInt("Denominatore: ");
            frazioni[pos] = new Frazione(num,den);
        }

        //fase di scrittura
        for (int pos = 0; pos < frazioni.length; pos++)
            out.println(frazioni[pos].toString());
    }
}
```

All'inizio viene creato un array di 10 elementi (costante MAX). Nel primo ciclo for sono letti il numeratore e il denominatore della frazione. Viene quindi creato l'oggetto frazione corrispondente e il suo riferimento è memorizzato nella posizione pos dell'array. Il secondo ciclo for scandisce invece gli elementi dell'array, uno alla volta, e li stampa invocando il metodo `toString`. Si noti che nell'ultima istruzione del metodo `main` il messaggio `toString` è inviato a `frazioni[pos]`, che è un riferimento a un oggetto di tipo `Frazione`.

I cicli for dell'esempio precedente mettono in evidenza la relazione che intercorre fra essi e gli array. I cicli for con questa struttura consentono infatti di scorrere le posizioni dell'array tramite un contatore e di compiere, all'interno del ciclo, un'operazione per ogni elemento dell'array.

Un aspetto importante degli array è che, una volta creati, non è più possibile modificarne la dimensione. Nell'esempio precedente, dopo aver creato `frazioni` come un array di MAX

posizioni non possiamo più cambiarne la dimensione. Però potremmo ovviamente riassegnare alla variabile `frazioni` un altro array di dimensione differente. D'altra parte non è necessario stabilire la dimensione dell'array quando scriviamo il codice: essa può essere decisa al momento dell'esecuzione. Ad esempio, potremmo sostituire la linea:

```
final int MAX = 10;
```

in cui viene fissato il numero di elementi dell'array, con la dichiarazione:

```
final int MAX = in.readInt("Quante frazioni vuoi inserire? ");
```

In questo modo sarà l'utente a decidere, in fase di esecuzione, il numero di frazioni che intende inserire nella sequenza.

Inizializzazione degli array

Un aspetto molto delicato dell'uso degli array riguarda la loro inizializzazione. Abbiamo osservato in precedenza che quando si tenta di utilizzare il valore di una variabile prima che sia stato effettuato un assegnamento a essa il compilatore segnala un errore. Ad esempio, nell'applicazione:

```
import prog.io.*;

class UsoErratoArray {

    public static void main(String[] args) {
        //predisposizione del canale di output
        ConsoleOutputManager out = new ConsoleOutputManager();

        String[] nomi;

        //fase di scrittura
        for (int pos = 0; pos < nomi.length; pos++)
            out.println(nomi[pos].toString());
    }
}
```

la variabile `nomi` viene utilizzata nell'intestazione del ciclo `for` senza che le sia stato assegnato un valore. Il compilatore non compila l'applicazione e segnala il seguente errore:

```
> javac UsoErratoArray.java
UsoErratoArray.java:12: variable nomi might not have been initialized
    for (int pos = 0; pos < nomi.length; pos++)
    ^
1 error
```

Ora invece consideriamo la stessa applicazione in cui correggiamo l'errore precedente:

```
import prog.io.*;
class UsoErratoArray {
    public static void main(String[] args) {
        //predisposizione del canale di output
        ConsoleOutputManager out = new ConsoleOutputManager();
        String[] nomi = new String[4];
        //fase di scrittura
        for (int pos = 0; pos < nomi.length; pos++)
            out.println(nomi[pos].toString());
    }
}
```

In questo caso abbiamo assegnato un valore alla variabile `nomi` e l'applicazione è correttamente compilata. Non avendo assegnato oggetti di tipo `String` alle varie posizioni dell'array, dobbiamo chiederci quali valori contengano. In effetti, in fase di creazione, le componenti dell'array vengono automaticamente inizializzate a un valore di default. Nel caso di array di oggetti tale valore è `null`. La situazione in memoria dopo l'esecuzione della dichiarazione

```
String[] nomi = new String[4];
```

è descritta nella Figura 5.2.

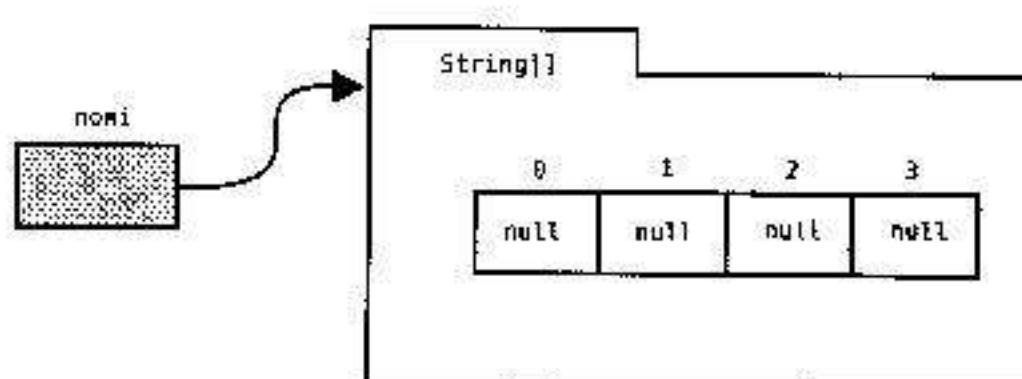


Figura 5.2 Array di String con le posizioni inizializzate a null.

Dato che sappiamo già che l'invocazione di un metodo su una variabile riferimento avente come valore `null` causa un errore in fase di esecuzione, non deve stupirci il risultato dell'esecuzione della precedente applicazione:

```
> java UsoErratoArray
Exception in thread "main" java.lang.NullPointerException
at UsoErratoArray.main(UsoErratoArray.java:13)
```

Si osservi che in questo caso l'errore si verifica quando la Java Virtual Machine trova l'invocazione del metodo `toString` inviata all'oggetto cui fa riferimento `nomi[pos]`. La variabile `nomi` risulta correttamente definita; infatti l'array è stato creato: `nomi.length` vale 4, ma tutte e quattro le posizioni contengono `null`.

Volendo inizializzare tutte le posizioni dell'array con la stringa vuota, al momento della definizione della variabile `nomi` possiamo scrivere:

```
String[] nomi = {"", "", "", ""};
```

ottenendo la situazione illustrata nella Figura 5.3.

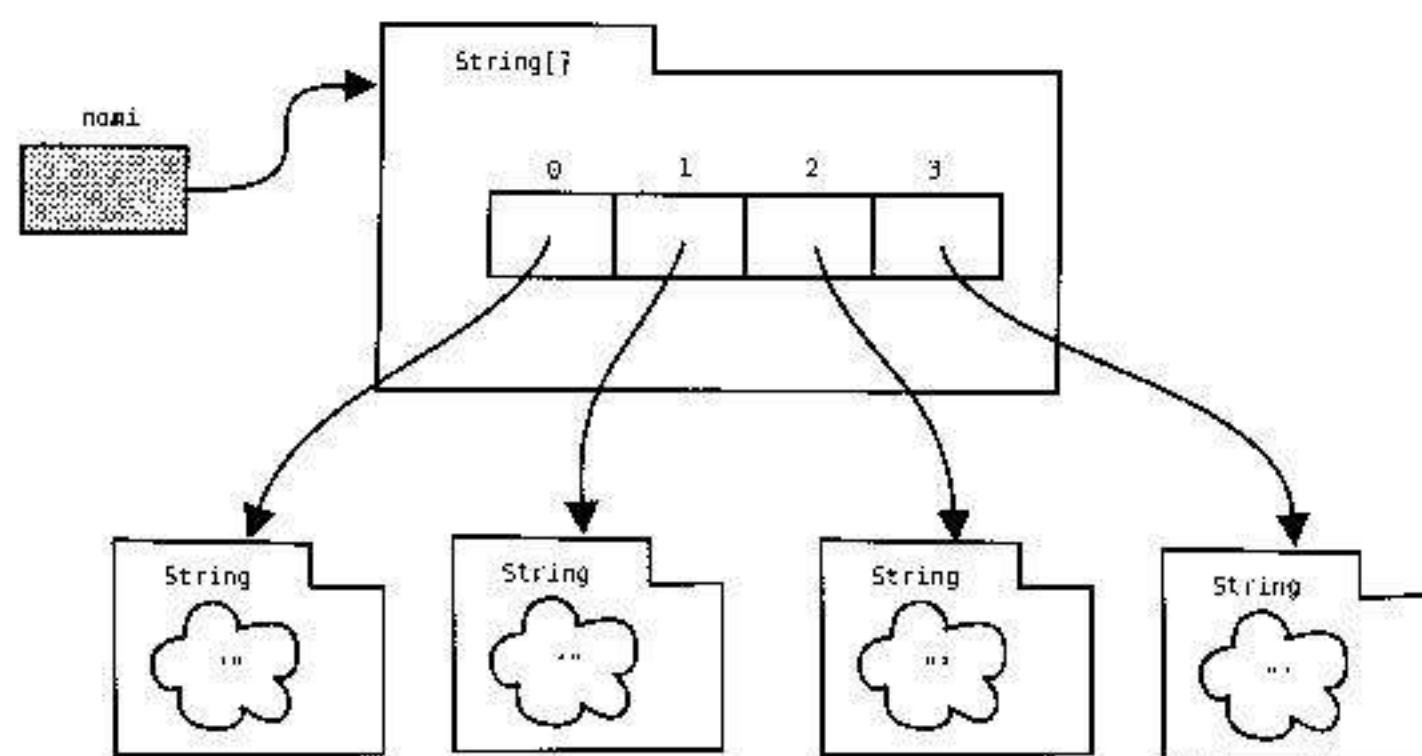


Figura 5.3 Array di stringhe vuote.

Nel caso in cui l'array abbia un grande numero di componenti, questo tipo di inizializzazione risulta molto scomoda. Se poi la dimensione dell'array è nota solo al momento dell'esecuzione, è impossibile effettuare un'inizializzazione di questo tipo. In questi casi possiamo però inizializzare tutte le posizioni dell'array con un semplice ciclo `for`. Ad esempio il codice

```
for (int pos = 0; pos < nomi.length; pos++)
    nomi[pos] = "";
```

dà luogo alla situazione descritta nella figura precedente.

Esempio: calcolo della frazione più vicina alla media

Come ulteriore esempio di uso degli array di oggetti scriviamo un'applicazione che, letta una sequenza di frazioni, ne calcola la media e individua, fra le frazioni lette, quella che le è più vicina. Per quel che riguarda la lettura delle frazioni procediamo come nell'esempio Sequenza-Frazioni, cioè chiediamo all'utente di indicare il numero MAX di frazioni che costituiranno la sequenza, costruiamo un array di MAX posizioni e quindi, all'interno di un ciclo `for`, procediamo alla lettura del numeratore e del denominatore, e costruiamo l'oggetto `Frazione` corrispondente

memorizzandolo nell'array. Per calcolare l'oggetto **Frazione** che rappresenta la media della sequenza inserita non sarebbe necessario conservare le frazioni della sequenza in un array: basterebbe calcolare la somma della sequenza, operazione che possiamo svolgere man mano che leggiamo la sequenza di frazioni e, terminata la fase di lettura, procedere a dividere tale somma per la frazione $\frac{MAX}{1}$. Riassumendo, la lettura e il calcolo della media sono realizzate dal seguente frammento di codice, dove `in` è un riferimento a un oggetto di tipo `ConsoleInputManager`:

```

Frazione[] frazioni = new Frazione[MAX];
Frazione somma = new Frazione(0);

// fase di lettura delle frazioni
for (int pos = 0; pos < frazioni.length; pos++) {
    out.println("Lettura della frazione " + (pos + 1));
    int num = in.readInt("Numeratore: ");
    int den = in.readInt("Denominatore: ");
    frazioni[pos] = new Frazione(num,den);
    somma = somma.piu(frazioni[pos]);
}
//calcolo della media;
int media = somma.diviso(new Frazione(MAX));

```

A questo punto vediamo come individuare nella sequenza il valore più vicino alla media. La distanza fra due frazioni può essere rappresentata da una frazione non negativa. Ad esempio la distanza fra le frazioni $\frac{1}{2}$ e $\frac{1}{3}$ è $\frac{1}{6}$. Ipotizziamo allora che `piùVicina` sia una variabile di tipo **Frazione**. All'inizio supponiamo che la frazione contenuta nella prima posizione dell'array sia quella più vicina alla media e la memorizziamo in `piùVicina`. Quindi, scorriamo l'array e, per ogni frazione in esso contenuta, verifichiamo se la nostra suposizione è corretta o no. Se lo è, passiamo a considerare la prossima frazione nell'array; se non lo è, rettifichiamo la nostra suposizione memorizzando in `piùVicina` la frazione che ha confutato la nostra ipotesi. Possiamo quindi schematizzare l'algoritmo come:

```

assegna a piùVicina la frazione in frazioni[0]
for (int pos = 1; pos < frazioni.length; pos++) {
    if (distanza fra frazioni[pos] e media è minore di quella
        fra piùVicina e media)
        memorizza frazioni[pos] in piùVicina
}

```

Per procedere all'implementazione dell'algoritmo dobbiamo anzitutto trovare un modo per calcolare la distanza fra due frazioni. Dato che la classe **Frazione** non fornisce un metodo per calcolare la distanza, dovremo ricorrere ai metodi `isMinore` e `meno`. Se `d1`, `d2` e `distanza` sono riferimenti a oggetti di tipo **Frazione**, il seguente frammento di codice calcola la distanza fra `d1` e `d2`:

```
if (d1.isMinore(d2))
```

```

distanza = d2.meno(d1);
else
    distanza = d1.meno(d2);

```

Poiché a ogni esecuzione del ciclo dovremo utilizzare la distanza fra la frazione memorizzata in `piùVicina` e quella contenuta nella posizione dell'array al momento considerata, durante l'esecuzione del ciclo è conveniente conservare la distanza fra la frazione attualmente memorizzata in `piùVicina` e quella memorizzata in `media`. A tale scopo introduciamo due variabili, `distanzaPiùVicina` e `distanzaCorrente`, e modifchiamo l'algoritmo nel modo seguente:

```

assegna a piùVicina la frazione in frazioni[0]
assegna a distanzaPiùVicina la distanza fra piùVicina e media
for (int pos = 1; pos < frazioni.length; pos++) {
    assegna a distanzaCorrente la distanza fra frazioni[pos] e media
    if (distanzaCorrente è minore di distanzaPiùVicina) {
        memorizza frazioni[pos] in piùVicina
        memorizza distanzaCorrente in distanzaPiùVicina
    }
}

```

Osserviamo che, essendo `distanzaPiùVicina` e `distanzaCorrente` riferimenti a oggetti di tipo `Frazione`, il test dev'essere effettuato mediante il metodo `isMinore`. Quindi ora possiamo scrivere il codice completo:

```

// individuazione della frazione più vicina alla media
Frazione piùVicina = frazioni[0], distanzaPiùVicina;

if (frazioni[0].isMinore(media))
    distanzaPiùVicina = media.meno(frazioni[0]);
else
    distanzaPiùVicina = frazioni[0].meno(media);

Frazione distanzaCorrente;
for (int pos = 1; pos < frazioni.length; pos++) {
    if (frazioni[pos].isMinore(media))
        distanzaCorrente = media.meno(frazioni[pos]);
    else
        distanzaCorrente = frazioni[pos].meno(media);

    if (distanzaCorrente.isMinore(distanzaPiùVicina)) {
        piùVicina = frazioni[pos];
        distanzaPiùVicina = distanzaCorrente;
    }
}

```

Il codice precedente può essere riscritto in modo più compatto osservando che anche il caso della frazione in `frazioni[0]` può essere trattato all'interno del ciclo.

```
// individuazione della frazione più vicina alla media
Frazione piùVicina = null, distanzaPiùVicina = null, distanzaCorrente;

for (int pos = 0; pos < frazioni.length; pos++) {
    if (frazioni[pos].isMinore(media))
        distanzaCorrente = media.meno(frazioni[pos]);
    else
        distanzaCorrente = frazioni[pos].meno(media);

    if (distanzaPiùVicina == null ||
        distanzaCorrente.isMinore(distanzaPiùVicina)) {
        piùVicina = frazioni[pos];
        distanzaPiùVicina = distanzaCorrente;
    }
}
```

In questo caso la variabile `distanzaPiùVicina` è inizializzata a `null` prima del ciclo. Alla prima iterazione, all'interno del ciclo viene calcolata la distanza fra la frazione `frazioni[0]` e `media`. Dato che `distanzaPiùVicina` vale `null`, la condizione del secondo `if` è verificata, e quindi a `piùVicina` viene associata la frazione in `frazioni[0]`. Perciò, dopo la prima iterazione del ciclo `for`, siamo nella stessa situazione del codice precedente, prima di iniziare a eseguire il ciclo `for`. Si osservi che nella condizione dell'ultimo `if` gioca un ruolo fondamentale il fatto che l'operatore `||` è valutato con la *lazy evaluation*. Se `distanzaPiùVicina == null` è vera allora la valutazione si interrompe. Se così non fosse, si avrebbe un errore in fase di esecuzione, in quanto nel secondo operando viene invocato il metodo `isMinore` fornendo come argomento `distanzaPiùVicina`.

Per concludere presentiamo l'intera applicazione:

```
import prog.io.*;
import prog.utili.Frazione;

class PiùVicinaAllaMedia {

    public static void main(String[] args) {
        // predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        // numero di frazioni da leggere
        final int MAX = in.readInt("Quante frazioni vuoi inserire ? ");
    }
}
```

```
if (MAX != 0) {
    Frazione[] frazioni = new Frazione[MAX];
    Frazione somma = new Frazione(0), media;

    // fase di lettura delle frazioni
    for (int pos = 0; pos < MAX; pos++) {
        out.println("Lettura della frazione " + (pos + 1));
        int num = in.readInt("Numeratore: ");
        int den = in.readInt("Denominatore: ");
        frazioni[pos] = new Frazione(num, den);
        somma = somma.piu(frazioni[pos]);
    }

    // calcolo della media
    media = somma.diviso(new Frazione(MAX));

    // individuazione della frazione più vicina alla media
    Frazione piùVicina = null, distanzaPiùVicina = null,
    distanzaCorrente;

    for (int pos = 0; pos < frazioni.length ; pos++) {
        if (frazioni[pos].isMinore(media))
            distanzaCorrente = media.meno(frazioni[pos]);
        else
            distanzaCorrente = frazioni[pos].meno(media);

        if (distanzaPiùVicina == null ||
            distanzaCorrente.isMinore(distanzaPiùVicina)) {
            piùVicina = frazioni[pos];
            distanzaPiùVicina = distanzaCorrente;
        }
    }

    // fase di scrittura
    out.println("La media è: " + media.toString());
    out.println("La frazione più vicina alla media è: " +
               piùVicina);
    out.println("La sua distanza dalla media è: " + distanzaPiùVicina);
}
}
```

5.2 Array e cicli for

Abbiamo già osservato la stretta relazione tra array e cicli `for`: ogni volta che si debbano esaminare tutte le posizioni di un array, dalla prima all'ultima, si può utilizzare un ciclo `for`. Ad esempio, se `a` è un array di oggetti di tipo `Frazione` e vogliamo visualizzare tutti gli elementi in esso contenuti, possiamo scrivere:

```
for (int i = 0; i < a.length; i++)
    out.println(a[i]);
```

In questo caso, è possibile produrre lo stesso effetto utilizzando la seguente forma abbreviata, detta anche *for-each* (dall'inglese “for each”, in italiano “per ogni”) nella quale non occorre introdurre esplicitamente l'indice `i`, ma una variabile del tipo base, in questo caso `Frazione`, alla quale saranno assegnati uno alla volta gli elementi contenuti nell'array, cioè riferimenti ad oggetti di tipo `Frazione`.

```
for (Frazione f : a)
    out.println(f)
```

In altre parole, il ciclo precedente è un'abbreviazione di:

```
for (int i = 0; i < a.length; i++) {
    Frazione f = a[i];
    out.println(f);
}
```

Esercizi

- 5.1 La seguente applicazione visualizza, una alla volta, le stringhe `cane`, `gatto` ed `elefante` e, per ciascuna di esse, chiede una nuova stringa.

```
import prog.io.*;
class ArrayString {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        String[] a = {"cane", "gatto", "elefante"};
        //lettura
```

```

        for (String s : a) {
            out.println("Stringa: " + s);
            s = in.readLine("Nuova stringa? ");
            out.println("E' stata inserita la stringa " + s);
        }

        //scrittura
        out.println("Contenuto dell'array: ");
        for (String s : a)
            out.println(s);
    }
}

```

Supponete di inserire, come nuove stringhe, `coniglio`, `pipistrello` e `tartaruga`. Cosa vi aspettate che venga stampato nel ciclo finale di visualizzazione? Confrontate la vostra risposta con ciò che viene prodotto dalla macchina facendo eseguire l'applicazione e spiegate la ragione del risultato ottenuto.

- 5.2 Riscrivete l'applicazione `PiùVicinaAllaMedia` sostituendo, ove possibile, i cicli `for` con cicli `for-each`.

5.3 Il parametro del metodo `main`

Dopo aver esaminato i metodi statici e gli array di oggetti siamo in grado di analizzare in modo dettagliato l'intestazione del metodo `main`, cioè:

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

che stiamo utilizzando nelle applicazioni. Come abbiamo visto in precedenza, `static` indica che si tratta di un metodo statico, e dunque invocabile anche se non esiste alcuna istanza della classe cui appartiene. L'argomento del metodo, di nome `args`, è un array di `String`: il metodo non restituisce alcun valore, in quanto il tipo restituito è `void`. Infine la parola chiave `public` è un modificatore di visibilità che indica che il metodo può essere invocato da chiunque sia in grado di accedere al codice della classe (esamineremo meglio il ruolo dei modificatori di visibilità più avanti).

Quando si manda in esecuzione una classe con il comando `java`, la Java Virtual Machine cerca nel bytecode della classe un metodo statico con il prototipo descritto sopra. Se lo trova, lo invoca passandogli come argomento l'array contenente le stringhe che sono state specificate nel comando di esecuzione dopo il nome della classe.

Ad esempio, se la classe `C` (già compilata) contiene un metodo `main`, il comando

```
java C pippo pluto
```

manda in esecuzione la classe C inizializzando l'array parametro di `main` con le stringhe "pioppo" e "pluto". Pertanto il campo `length` dell'array riferito da `args` contiene il numero di argomenti forniti sulla linea di comando. Se non sono stati forniti argomenti, si avrà un array vuoto, cioè di zero elementi. Si osservi che un array vuoto è diverso da un array (o, meglio, da un riferimento a un array) `null`.

Come esempio presentiamo una classe che riporta, uno per riga, gli argomenti forniti sulla riga di comando quando viene richiesta l'esecuzione:

```
import prog.io.*;
class Ripeti {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleOutputManager out = new ConsoleOutputManager();
        for (String s : args)
            out.println(s);
    }
}
```

Osserviamo che il metodo `main` non può essere un metodo statico, perché quando l'esecuzione inizia non esistono oggetti cui inviare messaggi.

5.4 Array di tipo primitivo

Come avevamo preannunciato, è possibile definire array di tipo primitivo. Anche questi array sono oggetti, e vengono dichiarati, creati e utilizzati nello stesso modo degli array di oggetti studiati prima. Ad esempio:

```
char[] iniziali;
int[] numeri;
```

definiscono, rispettivamente, `iniziali` e `numeri` come variabili riferimento a un array di `char` e a un array di `int`, mentre le espressioni

```
iniziali = new char[4]
numeri = new int[100]
```

assegnano rispettivamente alla variabile `iniziali` il riferimento a un array di `char` con 4 posizioni e alla variabile `numeri` un riferimento a un array di `int` con 100 posizioni. La differenza fra gli array di oggetti e gli array di tipo primitivo è analoga a quella che esiste fra le variabili riferimento e quelle di tipo primitivo. Come evidenziato nella Figura 5.4, mentre le posizioni di

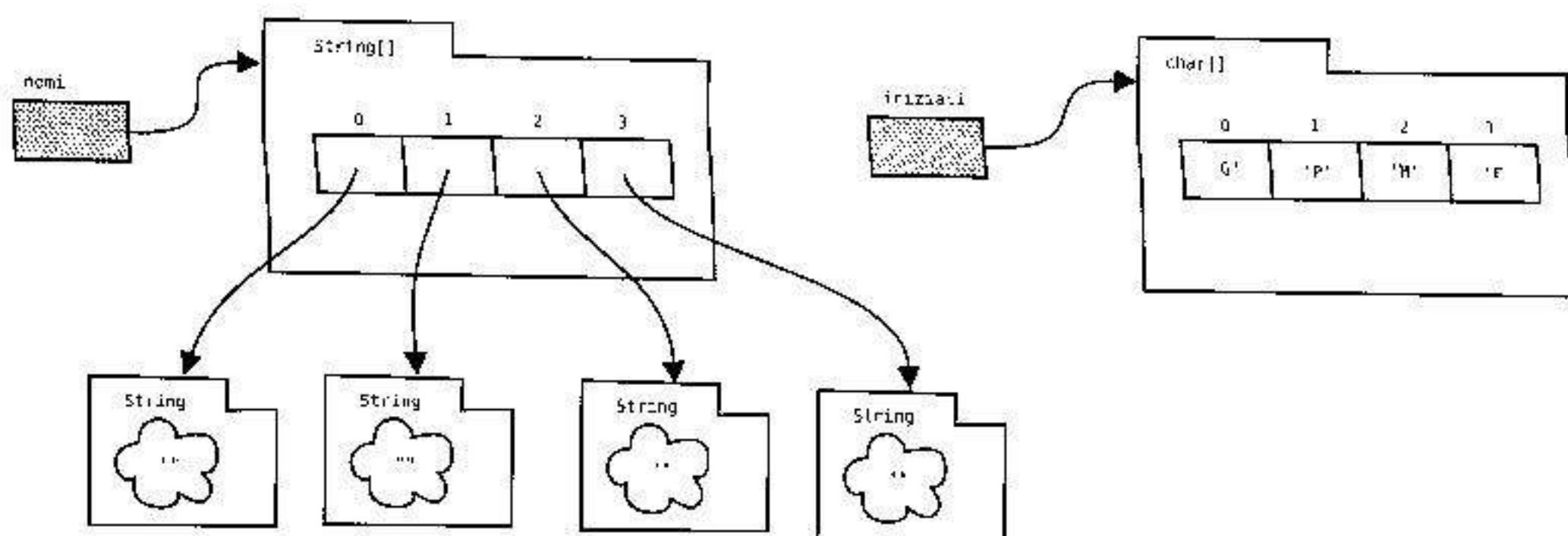


Figura 5.4 Array di riferimenti e array di char.

un array di oggetti, in questo caso un array di `String`, contengono riferimenti, le posizioni di un array di tipo primitivo contengono direttamente i valori.

Anche gli array di tipo primitivo dispongono di un campo `length` che contiene il numero di elementi dell'array. Per quel che riguarda l'inizializzazione è possibile specificare i valori iniziali dell'array (e implicitamente la sua dimensione) in fase di dichiarazione indicando fra parentesi graffe la sequenza dei valori da inserire nell'array. Ad esempio:

```
char[] iniziali = {'G', 'P', 'M', 'F'};
```

crea l'array con quattro posizioni rappresentato nella Figura 5.4.

Esempio: lettura di una sequenza di interi

Questa classe contiene un metodo `main` che legge alcuni numeri interi e li riscrive in uscita:

```
import prog.io.*;
class SequenzaInteri {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        final int MAX = 10;
        int[] tabella = new int[MAX];

        //fase lettura
```

```

int pos = 0;
int x = in.readInt();
while (x != 0) {
    tabella[pos] = x;
    pos++;
    x = in.readInt();
}

//fase di scrittura
for (int i = 0; i < pos; i++)
    out.println(tabella[i]);
}
}

```

All'inizio viene creato un array di 10 elementi (costante MAX). Nel ciclo while si leggono i numeri, che vengono caricati in posizioni successive dell'array. Lo zero è utilizzato per indicare la fine della sequenza, che viene poi scritta sul video.

Si osservi che se l'utente inserisce 11 numeri diversi da zero, il programma tenterà di accedere a una posizione inesistente dell'array. Questo provoca un errore in esecuzione (sperimentate questa situazione per vedere il messaggio d'errore prodotto). Possiamo modificare il ciclo di lettura, in modo da provocare l'uscita dal ciclo quando si raggiunge la fine dell'array:

```

import prog.io.*;

class SequenzaInteri {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        final int MAX = 10;
        int[] tabella = new int[MAX];

        //fase lettura
        int pos = 0;
        int x = in.readInt();
        while (x != 0 && pos < MAX) {
            tabella[pos] = x;
            pos++;
            if (pos < MAX)
                x = in.readInt();
        }
    }
}

```

```

    //fase di scrittura
    for (int i = 0; i < pos; i++)
        out.println(tabella[i]);
}
}

```

Una volta che l'array è stato creato, non è più possibile modificarne la dimensione. È possibile però creare un nuovo array e assegnarlo alla stessa variabile. Con questa tecnica si potrebbe modificare l'applicazione SequenzaInteri in modo da non avere un numero massimo di elementi fissato (si veda l'Esercizio 5.6). Questo impiego degli array è poco efficiente. In molte situazioni conviene dare la preferenza ad altre strutture dati.

Esempio: il crivello di Eratostene

Vogliamo determinare tutti i numeri primi minori o uguali a un numero ricevuto da input. A tal fine utilizziamo una tecnica detta *crivello di Eratostene*. L'idea fondamentale è molto semplice: dato un numero primo, tutti i suoi multipli non sono primi. Inizialmente l'algoritmo considera tutti i numeri maggiori o uguali a 2 come primi. Ad esempio, se stiamo considerando i numeri fino a 25, all'inizio abbiamo:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

In un secondo tempo si eliminano dall'elenco tutti i multipli del primo numero dell'elenco, cioè 2, ottenendo:

2 3 5 7 9 11 13 15 17 19 21 23 25

Si ripete il procedimento passando al numero successivo nell'elenco, cioè a 3, ed eliminandone i multipli:

2 3 5 7 11 13 17 19 23 25

Si passa poi a eliminare i multipli di 5:

2 3 5 7 11 13 17 19 23

e così via, fino alla fine dell'elenco (in realtà, osservando che se un numero ha un divisore, allora ha anche un divisore minore della propria radice quadrata, potremmo fermarci qui).

L'algoritmo può essere scritto utilizzando il seguente schema:

```

leggi nMax, il massimo numero da considerare
primi = elenco di tutti gli interi da 2 a nMax
for (numero = 2; numero <= nMax; numero++)
    if (numero e' nell'elenco dei primi)
        elimina tutti i multipli di numero dall'elenco dei numeri primi

```

I multipli di numero possono essere determinati mediante un ciclo in cui si comincia a considerare numero * 2 finché non si supera nMax:

```
for (multiplo = numero * 2; multiplo <= nMax; multiplo += numero)
    elimina multiplo dall'elenco dei numeri primi
```

Dopo questo procedimento è sufficiente visualizzare i numeri rimasti nell'elenco dei primi.

L'elenco dei numeri primi può essere realizzato con un array di boolean. Per comodità, dopo avere letto nMax, creiamo un array con nMax + 1 elementi e ne assegniamo il riferimento alla variabile primi. In questo modo nMax[x] è la componente associata all'intero x. In particolare nMax[x] conterrà true se e solo se x appartiene all'elenco dei primi. Le posizioni nMax[0] e nMax[1] non sono utilizzate.

Inizialmente tutti i numeri sono considerati primi. Pertanto tutti gli elementi dell'array vengono inizializzati a true. Successivamente, per eliminare un numero dall'elenco dei primi, è sufficiente porre a false la componente dell'array corrispondente. Infine si visualizzano i numeri la cui componente è true:

```
import prog.io.*;

class Eratostene {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        boolean[] primi;

        //lettura
        out.println("/** Calcolo numeri primi **");
        out.println();
        int nMax = in.readInt("Massimo numero da considerare? ");

        //creazione e inizializzazione array
        primi = new boolean[nMax + 1];
        for (int x = 2; x <= nMax; x++)
            primi[x] = true;

        for (int numero = 2; numero <= nMax; numero++)
            //se numero è primo allora tutti i suoi multipli non sono primi
            if (primi[numero])
                for (int multiplo = numero * 2; multiplo <= nMax;
                     multiplo += numero)
                    primi[multiplo] = false;
    }
}
```

```

//scrittura del risultato
out.println("I numeri primi fino a " + nMax + " sono:");
for (int numero = 2; numero <= nMax; numero++)
    //se il numero è primo allora visualizzalo
    if (primi[numero])
        out.println(numero);
}
}

```

Esercizi

- 5.3 Svolgete nuovamente l'Esercizio 4.21 utilizzando un array di 26 int per memorizzare le occorrenze delle lettere (una posizione per ciascuna lettera). In particolare l'elemento di posizione 0 dell'array verrà utilizzato per contare le 'a', quello di posizione 1 per contare le 'b', e così via. Osservate che se *x* è una variabile di tipo char che contiene una lettera minuscola, l'indice della posizione corrispondente nell'array è il risultato dell'espressione *x* - 'a' (il risultato prodotto da tale espressione è di tipo int).
- 5.4 Scrivete un'applicazione che, letta da tastiera una sequenza di numeri interi, dove il numero di elementi della sequenza è anch'esso stabilito dall'utente, produce in output la sequenza in ordine crescente (si suggerisce di memorizzare i numeri della sequenza in un array *e*, a partire da questo, costruire un altro array che contiene gli stessi elementi in ordine crescente).
- 5.5 Scrivete un'applicazione che, letta da tastiera una sequenza di frazioni, dove il numero di elementi della sequenza è stabilito dall'utente, produce in output la sequenza delle frazioni in ordine crescente.
- 5.6 Riscrivete l'applicazione *SequenzaInteri* in modo che tratti sequenze di lunghezza arbitraria. A tale scopo potete procedere come segue: inizialmente fate riferire *tabella* a un array di 10 elementi. Se durante il ciclo di lettura l'array viene riempito, create un nuovo array con un numero doppio di posizioni; nella prima metà copiate gli elementi già letti e contenuti nell'array riferito da *tabella*. Fate pertanto riferire *tabella* al nuovo array e proseguite l'inserimento. In altre parole, ogni volta che si satura la capacità dell'array, lo sostituite con uno nuovo, di capacità doppia, nel quale trasferite i dati.
- 5.7 Scrivete un'applicazione che legga una sequenza di interi e la riscriva in ordine crescente. La lunghezza della sequenza non deve essere fissata a priori; l'inserimento di zero segnala la fine della sequenza. Potete utilizzare le tecniche degli Esercizi 5.4 e 5.6.

5.5 Array e tipi enumerativi

Molto spesso i dati vengono rappresentati in tabelle in cui le posizioni sono individuabili mediante tipi enumerativi. In questo e nel paragrafo successivo, evidenzieremo come le tabelle siano implementabili in maniera del tutto naturale mediante gli array, e mostreremo l'uso di tipi enumerativi come indici per selezionare le posizioni degli array.

Supponiamo di dovere rappresentare le entrate mensili di una persona nell'arco di un anno. Una rappresentazione grafica naturale è una tabella come la seguente:

mese	entrate
Gennaio	1501,45
Febbraio	1472,97
Marzo	2314,03
Aprile	1499,85
Maggio	1708,22
Giugno	1335,44
Luglio	1881,18
Agosto	1454,22
Settembre	1234,32
Ottobre	1722,83
Novembre	1111,11
Dicembre	2722,72

o con una equivalente come

mese	Gennaio	Febbraio	...	Dicembre
entrate	1501,45	1472,97	...	2722,72

Nella prima colonna della tabella sono indicati i nomi dei mesi, nella seconda le entrate relative a ciascun mese. I nomi dei mesi sono fissati, mentre le entrate di ciascun mese potrebbero cambiare. La tabella non è altro che una funzione che associa a ogni mese (posizione della tabella) un importo. In particolare, vi sono dodici posizioni, ciascuna delle quali è in grado di contenere un importo. Possiamo rappresentare la tabella nella figura come un array formato da 12 oggetti di tipo `Importo`:³ l'oggetto in posizione 0 corrisponde all'importo di gennaio, quello in posizione 1 all'importo di febbraio e così via.

Una tabella di questo tipo potrebbe essere creata scrivendo:

```
Importo[] entrate = new Importo[NMESI];
```

dove la costante `NMESI` ovviamente vale 12. Per accedere all'importo corrispondente al mese di giugno potremmo scrivere:

```
entrate[5]
```

³ La classe `Importo` è stata presentata negli Esercizi 2.9 e 3.26.

Questa scrittura, sebbene corretta, risulta poco elegante e obbliga il programmatore a pensare in termini di rappresentazione (numeri interi) anziché di oggetti (mesi dell'anno). Il codice può essere reso più elegante utilizzando il tipo enumerativo `MeseDellAnno`, presentato nel Paragrafo 4.7: il mese di giugno è rappresentato dalla costante `MeseDellAnno.GIUGNO`. Non è possibile utilizzare direttamente le costanti del tipo enumerativo come selettori dell'array, ma è possibile utilizzare l'ordinale della costante enumerativa (ottenibile mediante il metodo `ordinal()`)

```
entrate[MeseDellAnno.GIUGNO.ordinal()]
```

La scrittura precedente, anche se poco sintetica, risulta maggiormente leggibile perché, grazie alla costante enumerativa, evidenzia il mese a cui ci si riferisce.

Supponiamo che l'array riferito dalla variabile `entrate` sia stato inizializzato inserendo degli importi per ciascuna posizione. Se si vuole visualizzare una tabella simile a quella riportata sopra si può scrivere un ciclo secondo il seguente schema:

```
for (ogni mese dell'anno) {
    visualizza il nome del mese
    visualizza le entrate del mese
}
```

La prima riga suggerisce l'uso di un ciclo `for-each`, in cui una variabile `mese` di tipo `MeseDellAnno` viene utilizzata per elencare tutti mesi. Per ciascuno di essi si visualizza il nome (metodo `toString` di `mese`) e le entrate, reperibili nella posizione `mese.ordinal()` dell'array:⁴

```
for (ogni mese dell'anno) {
    out.println(mese + " " + entrate[mese.ordinal()]);
}
```

I tipi enumerativi mettono a disposizione un metodo statico `values` utile per scrivere cicli `for-each` in cui a una variabile di controllo vengono assegnati, uno dopo l'altro, i valori del tipo enumerativo.⁵ Pertanto il ciclo può essere scritto come segue:

```
for (MeseDellAnno mese : MeseDellAnno.values()) {
    out.println(mese + " " + entrate[mese.ordinal()]);
}
```

Il ciclo precedente è molto più leggibile del seguente (in cui tra l'altro non riusciamo a risalire al nome del mese, ma solo al numero, a meno di complicare inutilmente il codice):

```
for (int mese = 0; i < 12; i++) {
    out.println(mese + " " + entrate[i]);
```

⁴ Come sempre, le chiamate dei metodi `toString()` sono implicite.

⁵ In realtà il metodo `values` restituisce un array contenente i riferimenti agli oggetti corrispondenti alle costanti che costituiscono il tipo enumerativo.

L'intestazione del primo ciclo, in particolare, evidenzia immediatamente che stiamo esaminando i mesi dell'anno; nel secondo ciò non è subito chiaro. Nel caso dei mesi il programmatore può essere facilmente tentato dall'uso dei numeri al posto delle costanti enumerative, in quanto l'associazione tra mese e numero è usuale (sebbene, qui si conti da 0, anziché da 1 come siamo abituati a fare). Per altri tipi non vi è alcuna associazione naturale. Supponete che un programmatore abbia definito un tipo `Colore`, enumerando alcuni colori. Volendo definire un array di interi, uno per ogni colore, si può procedere così:

```
final int NCOLORI = Colore.values().length;
int[] valori = new int[NCOLORI];
```

Nella prima istruzione viene calcolato il numero di costanti definite nel tipo enumerativo.⁶ Dopo che a ciascuna posizione dell'array sono stati assegnati dei valori, un ciclo che visualizzi il valore intero associato a ogni colore può essere scritto come:

```
for (Colore c : Colore.values()) {
    out.println(c + " " + valori[c.ordinal()]);
}
```

Questo frammento di codice è completamente svincolato dai dettagli relativi al tipo `Colore`. In particolare il programmatore non è obbligato a conoscere il valore intero corrispondente a ciascun colore. Inoltre, una modifica al tipo `Colore`, ad esempio con l'aggiunta di una nuova costante, non richiede alcuna modifica del codice.

Sviluppiamo ora una semplice applicazione che utilizza quanto discusso. A partire dalle entrate di ciascun mese di un anno (cioè da una tabella come quella iniziale), vogliamo calcolare la media mensile e giornaliera delle entrate di tutto l'anno. Infine, vogliamo calcolare, mese per mese, la media giornaliera delle entrate del mese.

L'applicazione sarà suddivisa in tre fasi. Nella prima fase, di acquisizione dei dati, viene creata la tabella delle entrate; nella seconda vengono calcolate e comunicate le medie relative a tutto l'anno; nell'ultima, infine, le medie giornaliere mese per mese.

L'applicazione memorizza i dati relativi alle entrate mensili in un array di oggetti di tipo `Importo`. In particolare, vengono preliminarmente effettuate le seguenti dichiarazioni e definizioni (per semplicità supponiamo l'anno non bisestile):

```
final int NMESI = MeseDellAnno.values().length; //12
final int NGIORNI = 365; //anno non bisestile
Importo[] entrate = new Importo[NMESI];
```

La fase di lettura può essere sviluppata secondo il seguente schema:

```
for (ogni mese) {
    leggi l'importo corrispondente alle entrate del mese
    memorizzalo nella posizione corrispondente di entrate
}
```

⁶ Abbiamo già osservato che il metodo statico `values` restituisce un array contenente i riferimenti agli oggetti associati alle costanti che costituiscono il tipo enumerativo. Accedendo al campo `length` possiamo sapere quanti sono.

La lettura dell'importo avviene chiedendo all'utente euro e centesimi e costruendo l'oggetto Importo corrispondente, che viene poi memorizzato nell'array.

```
for (MeseDellAnno mese : MeseDellAnno.values()) {
    //lettura e costruzione importo relativo al mese
    out.println("Entrate di " + mese + ":");
    int euro = in.readInt(" euro? ");
    int cent = in.readInt(" centesimi? ");
    Importo imp = new Importo(euro, cent);

    //memorizzazione importo
    entrate[mese.ordinal()] = imp;
}
```

Per la seconda fase, è necessario calcolare la somma delle entrate annue. A tale scopo si sommano tutti gli importi contenuti nella tabella. Possiamo definire una variabile somma di tipo Importo, inizializzata a zero, o più precisamente a un oggetto che rappresenta l'importo zero, a cui aggiungiamo via via gli elementi presenti nell'array:

```
Importo somma = new Importo(0);
for (MeseDellAnno mese : MeseDellAnno.values()) {
    Importo imp = entrate[mese.ordinal()];
    somma = somma.piu(imp);
}
```

Il codice precedente può essere semplificato: l'obiettivo è sommare tutti gli importi presenti nell'array; l'informazione relativa al mese è in questo caso irrilevante. Riscriviamo il codice con un ciclo for-each, che scandisce direttamente gli importi contenuti nell'array:

```
Importo somma = new Importo(0);
for (Importo imp : entrate)
    somma = somma.piu(imp);
```

A questo punto le medie possono essere calcolate dividendo la somma per il numero dei mesi e per il numero dei giorni dell'anno:

```
Importo mediaMensile = somma.diviso(NMESI);
Importo mediaGiornaliera = somma.diviso(NGIORNI);
```

I risultati così ottenuti verranno visualizzati con le usuali istruzioni.

Sviluppiamo ora la fase finale, nella quale, per ciascun mese, comunichiamo le entrate medie giornaliere. L'enunciazione del problema suggerisce già un ciclo for-each:

```
for (ogni mese)
    calcola e comunica le entrate medie giornaliere
```

Le entrate del mese sono disponibili nell'array e possono essere ottenute scrivendo

```
Importo entrataMese = entrate[mese.ordinal()];
```

La classe `MeseDellAnno` fornisce il metodo `numeroGiorni` che restituisce il numero di giorni del mese. Pertanto possiamo scrivere:

```
int giorniMese = mese.numeroGiorni();
```

Ricorrendo infine al metodo `diviso` della classe `Importo` si ottiene la media da visualizzare. Ecco la codifica di questa parte:

```
for (MeseDellAnno mese : MeseDellAnno.values()) {
    Importo entrataMese = entrate[mese.ordinal()];
    int giorniMese = mese.numeroGiorni();
    Importo mediaGiornalieraMese = entrataMese.diviso(giorniMese);
    out.print(mese + ": EURO ");
    out.println(mediaGiornalieraMese);
}
```

Segue il codice completo dell'applicazione:

```
import prog.io.*;
import prog.utili.Importo;
import prog.utili.MeseDellAnno;

class Entrate {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        final int NMESI = MeseDellAnno.values().length; //12
        final int NGIORNI = 365; //anno non bisestile
        Importo[] entrate = new Importo[NMESI];

        //fase di lettura
        for (MeseDellAnno mese : MeseDellAnno.values()) {
            //lettura e costruzione importo relativo al mese
            out.println("Entrate di " + mese + ":");
            int euro = in.readInt(" euro? ");
            int cent = in.readInt(" centesimi? ");
            Importo imp = new Importo(euro, cent);
            entrate[mese.ordinal()] = imp;
        }
    }
}
```

```

//memorizzazione importo
entrate[mese.ordinal()] = imp;
}

//calcolo delle medie
Importo somma = new Importo(0);
for (Importo imp : entrate)
    somma = somma.piu(imp);
Importo mediaMensile = somma.diviso(NMESI);
Importo mediaGiornaliera = somma.diviso(NGIORNI);

out.println("Entrate medie mensili: EURO " + mediaMensile);
out.println("Entrate medie giornaliere: EURO " + mediaGiornaliera);
out.println();

//calcolo e visualizzazione delle medie giornaliere
//nei singoli mesi
out.println("Entrate medie giornaliere per ciascun mese:");
for (MeseDellAnno mese : MeseDellAnno.values()) {
    Importo entrataMese = entrate[mese.ordinal()];
    int giorniMese = mese.numeroGiorni();
    Importo mediaGiornalieraMese = entrataMese.diviso(giorniMese);
    out.print(mese + ": EURO ");
    out.println(mediaGiornalieraMese);
}
}
}

```

Nell'applicazione precedente abbiamo dichiarato due volte una variabile di nome `imp`. Questo uso non crea alcun problema, in quanto le due dichiarazioni sono locali all'interno di due blocchi di codice (porzioni racchiuse tra graffe) distinte. Inoltre è piuttosto comodo, in quanto evita di inventare differenti nomi per variabili che, in realtà, hanno funzioni analoghe.

Si utilizzano spesso tabelle più complicate di quelle presentate qui. Supponiamo ad esempio, di volere aggiungere alla tabella dell'esempio iniziale, una colonna per le uscite. Una possibile rappresentazione consiste nell'utilizzare due array “paralleli”, uno per le entrate, l'altro per le uscite. In altre situazioni è opportuno invece utilizzare gli array di array, oggetto del paragrafo successivo.

Esercizi

5.8 Riscrivete l'applicazione `Entrate` in modo che riceva in ingresso anche l'indicazione dell'anno considerato e tenga conto del fatto che esso sia bisestile oppure no.

- 5.9 Scrivete un'applicazione che legga le entrate e le uscite in ciascun mese dell'anno e costruisca una tabella con il saldo, negativo o positivo, di ciascun mese.
- 5.10 Scrivete un'applicazione che legga, per ogni mese dell'anno, il numero di giorni lavorati. L'applicazione dovrà poi comunicare, per ciascun mese, il numero di giorni di lavoro e di riposo in quel mese e dall'inizio dell'anno.

5.6 Array di array

L'applicazione `Entrate` sviluppata nel paragrafo precedente utilizza un array per memorizzare le entrate mensili relative a un anno solare. Vogliamo ora costruire un'applicazione che tratti le entrate mensili relative a più anni. L'applicazione dovrà svolgere le seguenti operazioni:

- leggere da input due numeri interi, corrispondenti al primo e all'ultimo anno da considerare;
- leggere le entrate di ciascun mese del periodo considerato;
- comunicare le entrate medie mensili di ciascun anno;
- comunicare, per ciascuno dei 12 mesi, la media delle entrate sugli anni considerati.

Una rappresentazione grafica naturale dei dati del problema è fornita dalla tabella seguente:

	<i>Gennaio</i>	<i>Febbraio</i>	...	<i>Dicembre</i>
2002	1501,45	1472,97	...	2722,72
2003	1941,33	1345,78	...	1899,65
2004	1477,65	1441,21	...	1951,72

Ogni riga della tabella rappresenta un anno ed è costituita da dodici elementi, uno per ogni mese: come nell'applicazione `Entrate` ciascuna riga può essere implementata mediante un array di importi. L'intera tabella a sua volta è un array le cui righe sono i singoli array di importi. In sostanza, la tabella è un *array di array* di oggetti di tipo `Importo`.

Se `NANNI` è una variabile `int` che contiene il numero degli anni che vogliamo considerare e `NMESI` il numero di mesi di ciascun anno (ovviamente 12), possiamo definire un riferimento a un array di array con la struttura indicata sopra, come segue:

```
Importo[][] entrate = new Importo[NANNI][NMESI];
```

La dichiarazione specifica che `entrate` è un riferimento a un array i cui elementi sono array contenenti oggetti di tipo `Importo`. L'invocazione del costruttore specifica che si vuole costruire un oggetto formato da `NANNI` array, ognuno dei quali è formato da `NMESI` elementi. Gli array di array, o *array bidimensionali*, costruiti in questo modo sono anche detti *matrici*. Le matrici sono appunto tabelle formate da righe e colonne: ogni elemento dell'array è individuato da un indice di riga e da un indice di colonna.

Ad esempio, se NANNI vale 3, la matrice a cui fa riferimento `entrate` è rappresentata nella Figura 5.5. L'elemento `entrate[0]` rappresenta la prima riga della matrice (è un array di oggetti di tipo `Importo`); l'elemento `entrate[0][0]` è il primo elemento della prima riga della matrice, l'elemento `entrate[2][11]` è l'ultimo dell'ultima riga. Il numero di righe è fornito dal campo `length` di `entrate`, cioè scrivendo `entrate.length`; il numero di colonne dal campo `length` di una qualsiasi riga, ad esempio scrivendo `entrate[0].length`.⁷

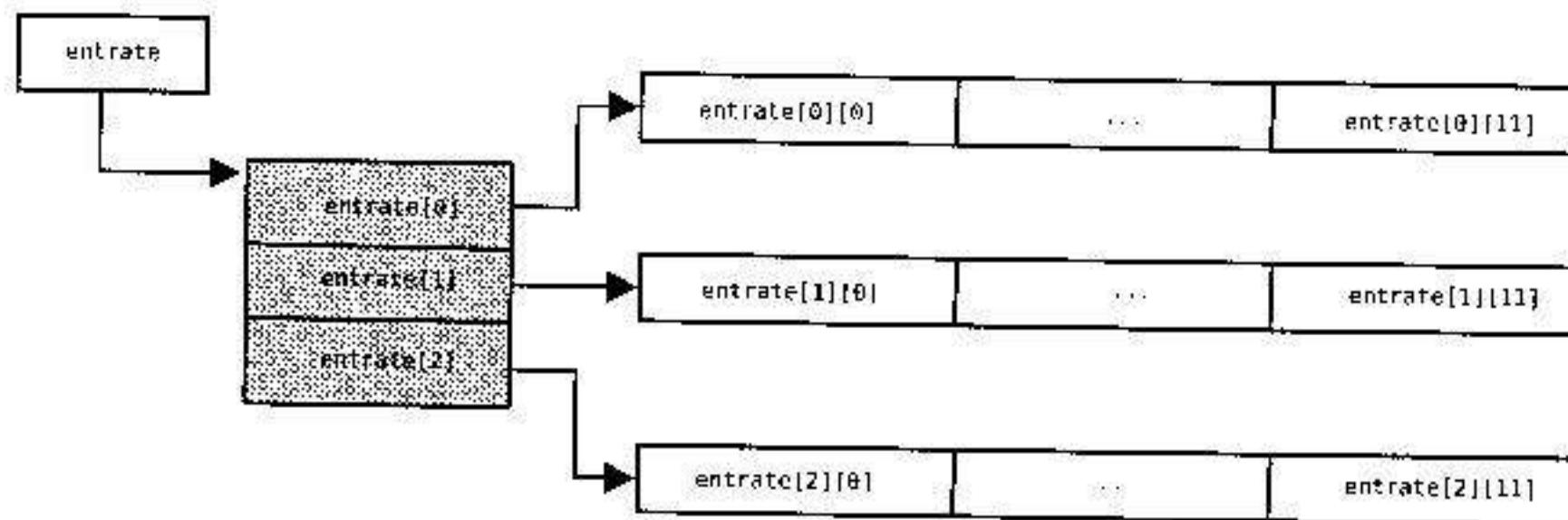


Figura 5.5 Rappresentazione della matrice `entrate`.

Tornando all'esempio che stiamo sviluppando e riferendoci alla tabella raffigurata all'inizio del paragrafo, è opportuno considerare come indice di riga l'anno e come indice di colonna il mese.

Iniziamo a scrivere la prima parte dell'applicazione, in cui viene richiesto all'utente di inserire il primo e l'ultimo anno da considerare. I due dati vengono memorizzati, rispettivamente, nelle variabili `primoAnno` e `ultimoAnno` di tipo `int`. L'applicazione, dopo avere controllato la correttezza dei dati, crea l'array per memorizzare i dati relativi alle entrate, richiamando il costruttore, come indicato in precedenza:

```

final int NMESI = MeseDellAnno.values().length; //12

//lettura intervallo anni
int primoAnno, ultimoAnno;
boolean intervalloScorretto;
do {
    primoAnno = in.readInt("Primo anno da considerare? ");
    ultimoAnno = in.readInt("Ultimo anno da considerare? ");
    if (primoAnno > ultimoAnno) {
        intervalloScorretto = true;
        out.println("Attenzione: l'ultimo anno non può precedere il primo!");
    } else
        intervalloScorretto = false;
} while (intervalloScorretto);
  
```

⁷ È anche possibile costruire *array frastagliati*, cioè array bidimensionali in cui le lunghezze delle righe sono diverse tra loro.

```
final int NANNI = ultimoAnno - primoAnno + 1;
Importo[][] entrate = new Importo[NANNI][NMESI];
```

Nella fase successiva, l'applicazione richiede all'utente di inserire le entrate di ciascuno dei mesi nell'intervallo considerato. La codifica può essere sviluppata secondo il seguente schema:

```
for (ogni anno nell'intervallo considerato)
    leggi e memorizza le entrate di ciascun mese dell'anno
```

L'istruzione interna al ciclo for può essere codificata con un ciclo, che scandisce ciascun mese:

```
for (ogni mese) {
    leggi le entrate relative al mese
    memorizzale nella posizione corrispondente all'anno e al mese
}
```

In sostanza, la codifica avviene con due cicli innestati: quello più esterno considera un anno alla volta, cioè una per una le righe della tabella; quello più interno considera, uno per uno, gli elementi di ciascuna riga, corrispondenti ai mesi.

Poiché gli indici delle posizioni degli array partono da zero, la riga di indice zero sarà quella corrispondente al primo anno considerato, la riga di indice 1 al secondo e così via. Per ottenere dal numero di un anno l'indice di riga corrispondente, è sufficiente sottrarre il contenuto della variabile `primoAnno`. Ecco la codifica di questa parte:

```
for (int anno = primoAnno; anno <= ultimoAnno; anno++)
    //lettura dati di anno
    for (MeseDellAnno mese : MeseDellAnno.values()) {
        //lettura e costruzione importo relativo al mese
        out.println("Entrate di " + mese + " " + anno + ":");
        int euro = in.readInt(" euro? ");
        int cent = in.readInt(" centesimi? ");
        Importo imp = new Importo(euro, cent);

        //memorizzazione importo
        entrate[anno - primoAnno][mese.ordinal()] = imp;
    }
```

Si osservi che il ciclo interno è pressoché identico al ciclo di lettura dell'applicazione `Entrate`. In particolare, si osservi che la tabella viene riempita riga per riga.

Sviluppiamo ora la parte di calcolo delle entrate medie di ciascun anno. Per ogni anno bisogna calcolare la somma delle entrate e dividerla per il numero dei mesi. In sostanza, si tratta di calcolare la media degli elementi contenuti in ciascuna riga della matrice. Anche in questo caso ricorriamo a due cicli innestati: in quello esterno si esamina un anno alla volta, cioè una per una le righe della matrice; nel ciclo interno, scorrendo via via gli elementi della riga che corrispondono ai mesi, si calcola la somma delle entrate. Terminato l'esame della riga si calcola e comunica la media delle entrate per quell'anno:

```

for (int anno = primoAnno; anno <= ultimoAnno; anno++) {
    Importo somma = new Importo(0);
    for (MeseDellAnno mese : MeseDellAnno.values())
        somma = somma.piu(entrata[anno - primoAnno][mese.ordinal()]);
    Importo media = somma.diviso(NMESI);
    out.print("Entrate medie mensili anno " + anno);
    out.println(": EURO " + media);
}

```

Consideriamo ora il problema del calcolo, per ciascuno dei dodici mesi, della media delle entrate rispetto ai diversi anni. In altre parole vogliamo calcolare la media delle entrate di gennaio, quella di febbraio, e così via. Per calcolare la media delle entrate di gennaio, dovremo calcolare la somma delle entrate di quel mese, nei diversi anni, e dividerla per il numero di anni considerati. La stessa operazione verrà ripetuta per tutti i dodici mesi. In altre parole, si tratta di calcolare la media degli elementi di ciascuna colonna della matrice. Ecco la codifica di questa parte:

```

for (MeseDellAnno mese : MeseDellAnno.values()) {
    Importo somma = new Importo(0);
    for (int anno = primoAnno; anno <= ultimoAnno; anno++)
        somma = somma.piu(entrata[anno - primoAnno][mese.ordinal()]);
    Importo media = somma.diviso(NANNI);
    out.print("Entrate medie nei mesi di " + mese);
    out.println(": EURO " + media);
}

```

Si noti la grande somiglianza tra questo codice e quello scritto in precedenza per il calcolo della media per ciascun anno: l'unica differenza sostanziale sta nei cicli. Nel codice che abbiamo scritto qui il ciclo esterno è sui mesi e quello interno sugli anni: questo ci permette di analizzare la tabella *per colonne*. Nel caso delle medie relative agli anni invece i cicli sono scambiati: l'analisi della tabella avviene *per righe*.

Ai fini del calcolo delle medie relative agli anni, i nomi dei mesi sono irrilevanti. Per questa ragione, il ciclo interno relativo ai mesi può essere sostituito da un ciclo for-each sugli elementi della riga corrispondente all'anno considerato, cioè sugli elementi dell'array `entrata[anno - primoAnno]`. La parte di calcolo e comunicazione delle medie relative agli anni può dunque essere riscritta come:

```

for (int anno = primoAnno; anno <= ultimoAnno; anno++) {
    Importo somma = new Importo(0);
    for (Importo imp : entrata[anno - primoAnno])
        somma = somma.piu(imp);
    Importo media = somma.diviso(NMESI);
    out.print("Entrate medie mensili anno " + anno);
    out.println(": EURO " + media);
}

```

Riportiamo il codice completo dell'applicazione così sviluppata:

```
import prog.io.*;
import prog.utili.Importo;
import prog.utili.MeseDellAnno;

class EntratePerAnni {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        final int NMESI = MeseDellAnno.values().length; //12

        //lettura intervallo anni
        int primoAnno, ultimoAnno;
        boolean intervalloScorretto;
        do {
            primoAnno = in.readInt("Primo anno da considerare? ");
            ultimoAnno = in.readInt("Ultimo anno da considerare? ");
            if (primoAnno > ultimoAnno) {
                intervalloScorretto = true;
                out.println("Attenzione: l'ultimo anno non può precedere il primo!");
            } else
                intervalloScorretto = false;
        } while (intervalloScorretto);

        final int NANNI = ultimoAnno - primoAnno + 1;

        Importo[][] entrate = new Importo[NANNI][NMESI];

        //fase di lettura
        for (int anno = primoAnno; anno <= ultimoAnno; anno++)
            //lettura dati di anno
            for (MeseDellAnno mese : MeseDellAnno.values()) {
                //lettura e costruzione importo relativo al mese
                out.println("Entrate di " + mese + " " + anno + ":");
                int euro = in.readInt(" euro? ");
                int cent = in.readInt(" centesimi? ");
                Importo imp = new Importo(euro, cent);
```

```

    //memorizzazione importo
    entrate[anno - primoAnno][mese.ordinal()] = imp;
}

//calcolo delle entrate medie di ogni anno
for (int anno = primoAnno; anno <= ultimoAnno; anno++) {
    Importo somma = new Importo(0);
    for (Importo imp : entrate[anno - primoAnno])
        somma = somma.piu(imp);
    Importo media = somma.diviso(NMESI);
    out.print("Entrate medie mensili anno " + anno);
    out.println(": EURO " + media);
}

//calcolo delle entrate medie di ogni mese
for (MeseDellAnno mese : MeseDellAnno.values()) {
    Importo somma = new Importo(0);
    for (int anno = primoAnno; anno <= ultimoAnno; anno++)
        somma = somma.piu(entrate[anno - primoAnno][mese.ordinal()]);
    Importo media = somma.diviso(NANNI);
    out.print("Entrate medie nei mesi di " + mese);
    out.println(": EURO " + media);
}
}
}

```

Esercizi

- 5.11 Modificate l'applicazione `EntratePerAnni` in modo che determini l'anno, tra quelli considerati, in cui vi sono state maggiori entrate.
- 5.12 Modificate l'applicazione `EntratePerAnni` in modo che determini tra tutti i mesi degli anni considerati, quello in cui vi sono state maggiori entrate (attenzione perché non si tratta di uno dei dodici mesi dell'anno, ma un mese di un anno tra quelli considerati; la risposta dovrà dunque indicare il nome di un mese e il relativo anno).
- 5.13 Modificate l'applicazione `EntratePerAnni` in modo che determini la media delle entrate mensili, relative a tutto il periodo considerato. L'applicazione dovrà poi elencare, anno per anno, i nomi dei mesi in cui le entrate sono state inferiori alla media calcolata.
- 5.14 Costruite un'applicazione che legga i dati relativi alle precipitazioni mensili che si sono verificate in un certo intervallo di anni e determini l'anno e il mese complessivamente più piovosi.

5.15 Costruite un'applicazione che riceva in ingresso i risultati di un referendum, espressi come numero di voti suddivisi tra *sì*, *no*, *schede bianche* e *schede nulle* e per aree geografiche (Nord, Centro, Sud e Isole), e produca in output:

- una tabella delle percentuali dei risultati nelle varie zone
- le percentuali complessive dei sì, no, schede bianche e schede nulle.

Per risolvere il problema potete utilizzare i tipi enumerativi `RispostaReferendum` e `AreaGeografica` presenti nel package `prog.utils` (nota: potete risolvere il problema senza nemmeno conoscere i nomi delle costanti presenti in questi tipi).

5.16 Per ognuno dei seguenti frammenti di codice individuate dichiarazioni di variabile ed eventualmente di tipo, in modo che le istruzioni che vi compaiono risultino corrette. Se ciò non fosse possibile, spiegatene il motivo.

- (1) `a = i > j;`
`x[i] = a;`
- (2) `a = i > j;`
`x[i] = 'a';`
- (3) `a = i > j;`
`x[i] = "a";`
- (4) `a = i > j;`
`x[i][j] = "a";`
`z = x[i];`
- (5) `q[q[i]] = i;`
- (6) `q[q[i]] = 'i';`
- (7) `q[i] = 'i';`
- (8) `q[i][i] = 'i';`
- (9) `q[i][i] = i;`

5.7 La classe Sequenza: introduzione ai tipi generici

Nel Capitolo 2 abbiamo presentato l'applicazione `Pappagallo`, il cui compito è quello di leggere una stringa e visualizzarla. L'applicazione utilizza un oggetto di tipo `String`, riferito dalla variabile `messaggio` in cui viene memorizzata la stringa letta. La fase di lettura (costituita da un'unica istruzione) ha l'effetto di produrre questo oggetto, utilizzato poi dalla fase di visualizzazione per mostrare la stringa sullo schermo.

Supponiamo ora di volere leggere una sequenza di stringhe e, una volta terminata la sequenza, di volerle visualizzare nell'ordine in cui sono state inserite. Supponiamo che la fine della sequenza venga indicata dall'utente premendo il tasto invio senza avere inserito alcun carattere, cioè inserendo la stringa vuota.

Analogamente all'applicazione Pappagallo, possiamo immaginare di svolgere questo compito in due fasi:

- nella fase di lettura, l'applicazione acquisisce la sequenza di stringhe che l'utente inserisce;
- nella successiva fase di visualizzazione, l'applicazione visualizza sul monitor tutte le stringhe della sequenza.

In questo caso, ciò che la fase di lettura produce, e che la fase successiva utilizza, è la sequenza di stringhe. Modelleremo questa sequenza con un apposito oggetto.

Prima di sviluppare quest'applicazione, introduciamo un problema simile, cioè quello di leggere una sequenza di frazioni e visualizzarla. Anche in questo caso possiamo adottare la stessa strategia indicata sopra, con un'unica differenza: il tipo di oggetti contenuti nella sequenza. Chiaramente possiamo presentare lo stesso problema in infinite versioni, semplicemente cambiando il tipo di oggetti che vogliamo trattare. In ogni caso, la strategia risolutiva potrà essere la stessa, basata su una sequenza di oggetti del tipo considerato. Per risolvere questa famiglia di problemi è pertanto utile disporre di una classe, che permetta di rappresentare sequenze di oggetti di un tipo prefissato. La classe **Sequenza**, definita nel package `prog.utili`, svolge proprio questo compito. Essa offre un costruttore privo di argomenti, che costruisce una sequenza vuota, un metodo `add` per aggiungere un elemento alla fine della sequenza, più altri metodi che descriveremo in seguito. Inoltre, è possibile scandire una sequenza, esaminando i suoi elementi uno alla volta, mediante un ciclo `for`.

Al fine di potere costruire sequenze di stringhe, di frazioni o di oggetti di un qualunque altro tipo, la classe **Sequenza** è *generica*: al momento dell'istanziazione della classe, cioè della creazione dell'oggetto sequenza, è necessario specificare il tipo degli oggetti che dovranno essere memorizzati nella sequenza stessa.⁸ Il tipo riferimento corrispondente a **Sequenza** viene detto *tipo generico*. Le classi e i tipi generici vengono indicati nella documentazione con una notazione del tipo **Sequenza<E>**, dove l'identificatore tra i simboli < e >, in questo caso E, viene detto *tipo parametro*. In sostanza, possiamo dire che la classe o il tipo è “**Sequenza di E**”. Al momento della creazione di un'istanza della classe, cioè di un oggetto **Sequenza** mediante l'invocazione del costruttore, occorrerà specificare il tipo effettivo degli oggetti che intendiamo inserire nella sequenza, cioè il *tipo argomento*. Ad esempio, volendo costruire un oggetto in grado di memorizzare una sequenza di stringhe, nell'invocazione del costruttore verrà specificato **String** come tipo argomento, scrivendo:

```
new Sequenza<String>()
```

Se invece vogliamo costruire una sequenza di frazioni, invocheremo il costruttore fornendo come tipo argomento **Frazione**:

```
new Sequenza<Frazione>()
```

⁸ In realtà dovremmo scrivere “è possibile specificare”, anziché “è necessario specificare”. I progettisti del linguaggio non hanno reso obbligatoria la specifica del tipo degli oggetti in maniera che il codice scritto per le versioni precedenti del linguaggio sia ancora utilizzabile. È bene però che il nuovo codice che si scrive contenga tale specifica.

Elenchiamo brevemente alcuni metodi offerti dalla classe `Sequenza<E>` (dove si legge `E` si intende il tipo argomento fornito al momento della creazione della sequenza):

- `public boolean add(E o)`
Aggiunge alla fine della sequenza l'oggetto fornito tramite l'argomento `o` e restituisce `true`.
Nel caso come argomento venga fornito `null`, non modifica la sequenza e restituisce `false`.⁹
- `public int size()`
Restituisce il numero di elementi presenti nella sequenza.
- `public boolean isEmpty()`
Restituisce `true` se e solo se la sequenza è vuota.
- `public boolean contains(E o)`
Restituisce `true` se e solo se la sequenza contiene un oggetto uguale (sulla base del criterio di uguaglianza fornito dal metodo `equals`) a quello specificato tramite l'argomento.
- `public E find(E o)`
Restituisce il riferimento al primo oggetto nella sequenza uguale a quello specificato tramite l'argomento, o `null` se tale oggetto non è presente.
- `public boolean remove(E o)`
Elimina dalla sequenza il primo oggetto uguale a quello specificato tramite l'argomento `o` e restituisce `true`. Nel caso tale oggetto non vi sia, lascia la sequenza immutata e restituisce `false`.

Scriviamo ora un'applicazione in grado di risolvere il problema iniziale, cioè quello di leggere una sequenza di stringhe e di visualizzarle. Chiameremo quest'applicazione `PappagalloConMemoria`. Prima della lettura dei dati, predisponiamo l'oggetto di tipo `Sequenza<String>`, memorizzandone il riferimento in una variabile di nome `memo`. Il tipo di tale variabile sarà `Sequenza<String>`:

```
Sequenza<String> memo = new Sequenza<String>();
```

Il tipo `Sequenza<String>` viene detto *tipo parametrizzato*.¹⁰ La fase di lettura è basata su un ciclo nel quale si legge una stringa: se essa non è vuota la si aggiunge alla sequenza (metodo `add`) e si passa quindi a leggere la stringa successiva:

```
String s = in.readLine();
while (!s.equals("")) {
```

⁹ Si osservi che in base a ciò che è stato sottolineato in precedenza, il metodo `add` di un oggetto di tipo `Sequenza<String>` riceverà un argomento di tipo `String`. Il tentativo di fornire un argomento di tipo diverso, come ad esempio `Frazione`, provocherà un errore in fase di compilazione.

¹⁰ Talvolta il tipo parametrizzato viene anche chiamato *invocazione* del tipo generico. Tuttavia preferiamo evitare questa terminologia che potrebbe creare confusione con le invocazioni dei costruttori e dei metodi.

```

    memo.add(s);
    s = in.readLine();
}

```

Nella fase di visualizzazione bisogna scorrere la sequenza, stampando le stringhe in essa contenute. Per scorrere gli elementi contenuti in un oggetto di tipo `Sequenza<E>`, dal primo all'ultimo, è possibile utilizzare, in analogia a quanto avviene per gli array, un ciclo `for-each` nel quale si dichiara una variabile di tipo `E` e si indica il riferimento alla sequenza. Ad esempio nel caso che stiamo esaminando possiamo scrivere:

```
for (String x : memo)
```

La variabile `x` "scorrerà" gli elementi della sequenza: alla prima iterazione si riferirà alla prima stringa contenuta nella sequenza, alla seconda iterazione alla seconda stringa, e così via, sino al raggiungimento della fine della sequenza e della conseguente terminazione del ciclo (chiaramente se la sequenza è vuota il ciclo termina subito). Per molte delle collezioni presenti nelle librerie è possibile utilizzare un ciclo con questa struttura. Forniremo maggiori dettagli più avanti nel testo. La fase di visualizzazione può essere realizzata semplicemente come:

```

for (String x : memo)
    out.println(x);

```

Segue il testo completo dell'applicazione così realizzata:

```

import prog.io.*;
import prog.utili.Sequenza;

class PappagalloConMemoria {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //predisposizione della "memoria"
        Sequenza<String> memo = new Sequenza<String>();

        //fase di lettura
        String s = in.readLine();
        while (!s.equals("")) {
            memo.add(s);
            s = in.readLine();
        }

        //fase di scrittura

```

```
    for (String x : memo)
        out.println(x);
}
```

Presentiamo ora l'analogo esempio relativo alle frazioni. Per memorizzare la sequenza utilizziamo una variabile di tipo `Sequenza<Frazione>`. La fase di lettura presenta alcune differenze rispetto all'esempio precedente. In particolare, a ogni passo chiediamo all'utente se intenda inserire un'altra frazione oppure no:

```
import prog.io.*;
import prog.utili.Frazione;
import prog.utili.Sequenza;

class PappagalloFrazioni {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //crea una sequenza inizialmente vuota
        Sequenza<Frazione> frazioni = new Sequenza<Frazione>();

        do {
            //leggi una frazione e costruisci l'oggetto corrispondente
            out.println("Inserisci la frazione:");
            int num = in.readInt(" -- numeratore? ");
            int den = in.readInt(" -- denominatore? ");
            Frazione f = new Frazione(num, den);

            //aggiungi la frazione alla sequenza
            frazioni.add(f);
        } while (in.readSiNo("Vuoi inserire altre frazioni? (s/n) "));

        out.println("Sono state inserite le seguenti frazioni");
        for (Frazione f : frazioni)
            out.println(f);
    }
}
```

Alcune osservazioni sulla compilazione

I tipi generici sono stati introdotti a partire dalla versione 5 del linguaggio Java. Per fare in modo che il codice scritto precedentemente sia ancora utilizzabile, il compilatore permette di servirsi di classi generiche, come `Sequenza`, senza specificare il tipo argomento. Ad esempio, nella chiamata del costruttore dell'applicazione `PappagalloConMemoria`, si potrebbe scrivere:

```
Sequenza memo = new Sequenza();
```

oppure

```
Sequenza memo = new Sequenza<String>();
```

o anche

```
Sequenza<String> memo = new Sequenza();
```

In nessuno dei tre casi l'istruzione provoca un errore durante la compilazione (nei primi due casi, in conseguenza di quest'istruzione, si ha un errore nella compilazione del successivo ciclo `for`), tuttavia il compilatore fornisce la seguente nota:

Note: `PappagalloConMemoria.java` uses unchecked or unsafe operations.

Note: Recompile with `-Xlint:unchecked` for details.

La nota indica che si stanno utilizzando tipi generici senza indicare il tipo argomento. In questo caso il codice potrebbe contenere errori relativi ai tipi che possono provocare malfunzionamenti in esecuzione. Per ottenere maggiori dettagli occorre ricompilare utilizzando l'opzione `-Xlint:unchecked` o, più semplicemente, `-Xlint`.

In ogni caso, tranne quando si debba fare riferimento a codice sviluppato per versioni precedenti del linguaggio, è opportuno indicare sempre il tipo argomento. Gli avvertimenti forniti dal compilatore risultano utili per correggere eventuali dimenticanze.

La classe SequenzaOrdinata

La libreria `prog.utili` offre anche la classe `SequenzaOrdinata<E>` le cui istanze sono sequenze ordinate di oggetti di un tipo `E`. Ad esempio, se `E` è il tipo `String`, le sequenze saranno ordinate secondo l'ordine alfabetico, se `E` è il tipo `Frazione` o il tipo `Integer` le sequenze saranno ordinate in maniera crescente, se `E` è il tipo `Data` le sequenze saranno ordinate cronologicamente. Anche `SequenzaOrdinata` dispone di un costruttore privo di argomenti che costruisce la sequenza vuota. I prototipi dei metodi sono gli stessi di `Sequenza`, così come i contratti, con la sola eccezione del contratto del metodo `add` che inserisce il nuovo elemento nella sequenza in maniera che essa resti ordinata.

Riportiamo a titolo di esempio il codice di un'applicazione che legge una sequenza di stringhe e la visualizza in ordine alfabetico. Si noti che l'unica differenza rispetto al codice dell'applicazione `PappagalloConMemoria` sta nell'uso di `SequenzaOrdinata` al posto di `Sequenza`.

```
import prog.io.*;
import prog.utili.SequenzaOrdinata;

class PappagalloOrdinato {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //predisposizione della "memoria"
        SequenzaOrdinata<String> memo = new SequenzaOrdinata<String>();

        //fase di lettura
        String s = in.readLine();
        while (!s.equals("")) {
            memo.add(s);
            s = in.readLine();
        }

        //fase di scrittura
        for (String x : memo)
            out.println(x);
    }
}
```

Mentre è possibile costruire sequenze di oggetti di un qualunque tipo, per costruire sequenze ordinate è necessario che il tipo sia “ordinabile”, cioè sia possibile definire una relazione di ordine totale tra i suoi elementi. Non tutti i tipi hanno questa caratteristica. Si consideri, ad esempio, un tipo **Soprammobile**: quale può essere una relazione di ordine “naturale” tra due soprammobili? Dunque, vi sono alcune limitazioni alla genericità della classe **SequenzaOrdinata** che descriveremo più avanti nel testo.

Esercizi

- 5.17 Servendovi della classe **Sequenza** scrivete un'applicazione che legga una sequenza di numeri interi e la ristampi. Si osservi che una sequenza è destinata a contenere oggetti, quindi è necessario ricorrere alla classe involucro **Integer**; si noti in questa situazione l'utilità dei meccanismi di unboxing e autoboxing.
- 5.18 Scrivete un'applicazione che legga una sequenza di numeri interi e ne calcoli la media. L'applicazione dovrà poi indicare quanti numeri, tra quelli letti, risultano superiori alla media e quanti inferiori.

5.19 Scrivete un'applicazione che legga una sequenza di frazioni e, successivamente, una frazione g. L'applicazione deve comunicare:

- la più grande frazione della sequenza che risulti minore di g;
- la più piccola frazione della sequenza che risulti maggiore di g.

5.20 Risolvete l'esercizio precedente leggendo *prima* la frazione g e *poi* la sequenza. In questo caso *non è* necessario memorizzare la sequenza delle frazioni.

5.21 Riscrivete l'applicazione PiuVicinaAllaMedia utilizzando la classe Sequenza al posto degli array. La nuova versione dell'applicazione non dovrà richiedere preliminarmente all'utente il numero di frazioni da inserire.

5.22 Risolvete l'esercizio precedente servendovi della classe SequenzaOrdinata.

5.23 Scrivete un'applicazione che legga un numero intero n e stampi, in ordine alfabetico, tutti i numeri da 1 a n scritti in lettere (utilizzate le classi SequenzaOrdinata, Intero e String).

5.24 Si consideri la classe Insieme<E> fornita nel package prog.utili. Le istanze di questa classe rappresentano insiemi di oggetti in senso insiemistico, cioè collezioni di oggetti che non contengono duplicazioni.

La classe fornisce un costruttore privo di argomenti per costruire un oggetto che rappresenta l'insieme vuoto. Vengono inoltre forniti, tra gli altri, i seguenti metodi.

- `public boolean add(E o)`
Aggiunge all'insieme l'oggetto specificato tramite l'argomento, se non già presente. Restituisce `true` se l'elemento è stato aggiunto, `false` altrimenti.
- `public int size()`
Restituisce il numero di elementi presenti nell'insieme.
- `public boolean isEmpty()`
Restituisce `true` se e solo se l'insieme è vuoto.
- `public boolean contains(E o)`
Restituisce `true` se e solo se l'insieme contiene un elemento uguale a quello specificato tramite l'argomento.
- `public boolean remove(E o)`
Elimina dall'insieme l'oggetto specificato tramite l'argomento, se presente, e restituisce `true`. Nel caso l'oggetto non sia presente, lascia l'insieme immutato e restituisce `false`.

Anche per gli insiemi, come per le sequenze, è possibile utilizzare il ciclo `for-each` per scandire uno alla volta gli elementi (tuttavia gli elementi non appaiono essere in un ordine particolare).

Riscrivete l'applicazione Eratostene servendovi delle classi Insieme e Integer, al posto degli array.

- 5.25 Utilizzando la classe `Insieme` e la classe `Character`, scrivete un'applicazione che legga due stringhe e visualizzi l'insieme dei caratteri che appaiono in almeno una delle due stringhe e un elenco dei caratteri che appaiono in entrambe le stringhe. Ecco come potrebbe essere un esempio di esecuzione:

```
Prima stringa? la pappa di pippo
Seconda stringa? la pipa di pluto
Insieme caratteri prima stringa: od iapl
Insieme caratteri seconda stringa: od iautpl
Insieme caratteri che compaiono nelle due stringhe: doiautpl
Insieme caratteri che compaiono in entrambe le stringhe: doiapl
```

Si suggerisce di costruire prima di tutto un insieme di caratteri per ciascuna stringa. Costruite poi l'unione e l'intersezione dei due insiemi.

- 5.26 Riscrivete l'applicazione richiesta per l'esercizio precedente in modo che i caratteri vengano visualizzati in ordine lessicografico (potete scandire gli elementi di un insieme con un ciclo `for-each` e costruire una `SequenzaOrdinata`).
- 5.27 Data una sequenza di valori x_1, \dots, x_n ordinati in modo crescente, la *mediana* M è l'elemento che occupa il posto centrale se il numero di elementi è dispari, altrimenti è la media dei due elementi centrali. In altre parole, la mediana è data da:

$$M = \begin{cases} x_{\frac{(n+1)}{2}} & \text{se } n \text{ è dispari} \\ \frac{x_{\frac{n}{2}} + x_{\frac{n}{2}+1}}{2} & \text{altrimenti} \end{cases}$$

La *moda* è invece il valore che compare il maggior numero di volte nella sequenza.

Scrivete un'applicazione che legga una sequenza di frazioni e ne visualizzi la media, la mediana e la moda.

- 5.28 Scrivete un'applicazione che legga una sequenza numeri e la visualizzi sul monitor:

- nello stesso ordine in cui è stata inserita,
- in ordine crescente,
- in lettere e in ordine alfabetico.

Supponete che l'inserimento dello zero indichi la fine della sequenza. Ad esempio, se vengono forniti in ingresso i valori 13, 472, 9, 1, 0, l'applicazione dovrà fornire il seguente output:

Numeri inseriti:

13

472

9
1

Ordine crescente:

1
9
13
472

Ordine alfabetico:

nove
quattrocentosettantadue
tredici
uno

Per risolvere l'esercizio, utilizzate le classi Sequenza, SequenzaOrdinata, e Intero del package prog.utili. Per ottenere la sequenza in ordine alfabetico, si suggerisce di scandire la sequenza di numeri letti, costruendo una sequenza ordinata di stringhe (servitevi del metodo `toString` della classe Intero per produrre le stringhe).

Uso della gerarchia

Nel Capitolo 2 abbiamo introdotto le nozioni basilari della programmazione a oggetti partendo dall'esempio di `PinoPasticcino`; in questo capitolo riprendiamo quell'esempio per presentare le nozioni di *ereditarietà* e *gerarchia*. Come avevamo osservato allora, e come abbiamo sperimentato utilizzando diversi tipi di classi nei capitoli successivi, le classi rappresentano categorie di oggetti caratterizzati dallo stesso tipo di informazione, stato, e dal medesimo comportamento. Un'istanza (un oggetto) della classe è in grado di rispondere ai messaggi definiti nella classe; la classe specifica il metodo, cioè l'insieme di operazioni da eseguire per rispondere a un messaggio.

Torniamo ora a considerare il caso della nostra pasticceria di fiducia. Dopo aver ordinato la torta ci aspettiamo che ci venga chiesto di pagarla e ci aspettiamo anche che ci vengano prospettate diverse opzioni di pagamento, ad esempio in contanti o con carta di credito. Ipotizziamo di voler pagare tramite carta di credito: in questo caso possiamo supporre che il commesso della pasticceria sia in grado di riconoscere il messaggio `pagamentoCC(numeroCarta)`. Ovviamente ogni pasticceria mette a disposizione questa possibilità, e quindi immaginiamo che questo metodo sia specificato nella classe `Pasticceria`. Come abbiamo osservato nei capitoli precedenti, quando un oggetto riceve un messaggio che compare nella sua interfaccia, reagisce a esso eseguendo la sequenza di operazioni (il codice) del metodo relativo. Nella Figura 6.1 è descritto ciò che accade quando `daPinoPasticcino` riceve il messaggio `pagamentoCC`. Nella figura la classe è rappresentata da un rettangolo che contiene il nome della classe e in cui compare l'elenco dei metodi che essa mette a disposizione (per comodità abbiamo indicato solo quelli che ci interessano in questo esempio). La notazione utilizzata, che è leggermente diversa da quella di Java e che descriveremo nei dettagli in questo capitolo, è quella di UML (*Unified Modeling Language*) per rappresentare classi e oggetti. Supponiamo che la Java Virtual Machine stia eseguendo un'applicazione che contiene le seguenti istruzioni

```
Pasticceria daPinoPasticcino = new Pasticceria();
...
daPinoPasticcino.pagamentoCC(numeroCarta);
```

Dato che le istruzioni (in bytecode) da eseguire in risposta a un messaggio sono memorizzate nella classe (nel file con l'estensione `.class`), per prima cosa la Java Virtual Machine deve

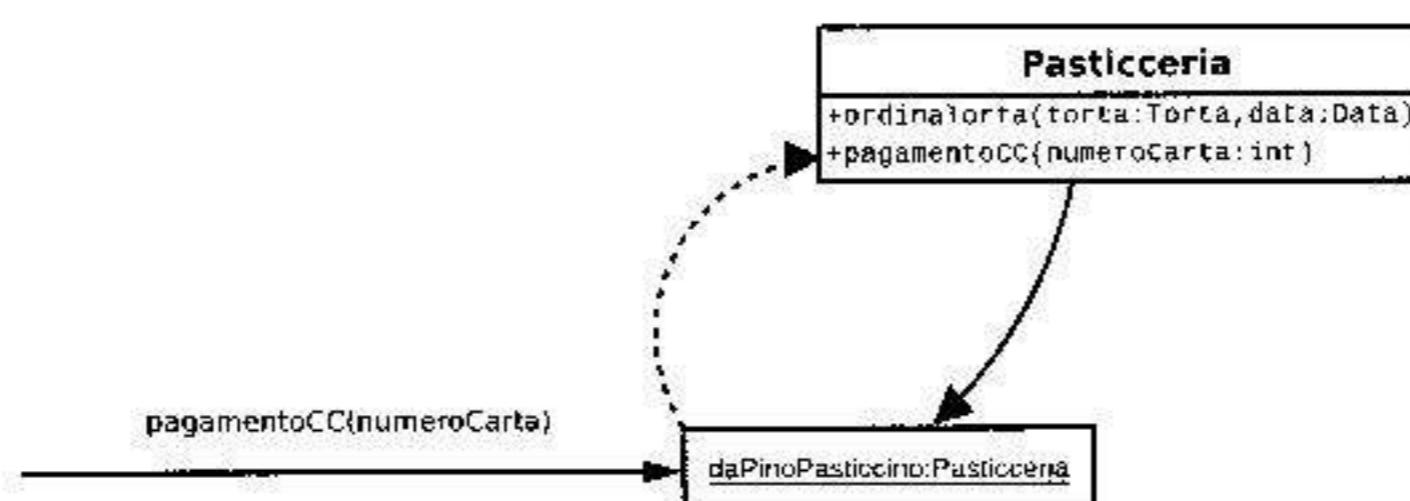


Figura 6.1 Esecuzione del metodo `pagamentoCC`.

recuperare il codice del metodo. L'informazione di cui ha bisogno per cercare il metodo è memorizzata nell'oggetto ed è costituita dal nome della classe che è stata utilizzata per crearlo. Dato che, nel nostro caso, `daPinoPasticcino` è un'istanza di `Pasticceria`, la Java Virtual Machine cerca il metodo `pagamentoCC`, che prevede un intero come argomento, nella classe `Pasticceria` (per la precisione lo cerca nel file `Pasticceria.class`). Recuperato il codice del metodo, questo viene eseguito dall'oggetto `daPinoPasticcino`.

Torniamo ora a un esempio della vita reale e supponiamo di entrare in una libreria per acquistare dei libri. Anche in questo caso possiamo pensare alla libreria come a un'istanza di una classe `Libreria` che specifica l'insieme dei messaggi accettati da una libreria e le sequenze di operazioni da eseguire in risposta a tali messaggi. Inoltre ci aspettiamo di poter effettuare un pagamento con la carta di credito, cioè che la classe `Libreria` metta anch'essa a disposizione un metodo con la segnatura `pagamentoCC(int numeroCarta)`. In realtà ci aspettiamo anche che il contratto e la sequenza di operazioni eseguite in risposta al messaggio siano le stesse dell'analogo metodo della classe `Pasticceria`. Se eseguiamo un'applicazione Java in cui compaiono le istruzioni

```

Libreria daManuzio = new Libreria();
...
daManuzio.pagamentoCC(numeroCarta);
  
```

come evidenziato nella Figura 6.2, la Java Virtual Machine procederà in maniera analoga all'esempio precedente andando a cercare il metodo da eseguire nella classe `Libreria`.

Se pensiamo più attentamente a quanto facciamo nella vita reale, ci accorgiamo che il fatto che `daManuzio` e `daPinoPasticcino` accettino il pagamento per mezzo di carta di credito, e quindi prevedano nella loro interfaccia il messaggio `pagamentoCC`, non è perché sono una libreria e una pasticceria, ma perché appartengono a una *categoria più generale*, la categoria dei negozianti, che rende disponibile tale messaggio. Come risulta chiaro dall'esempio, il nostro modo di organizzare la conoscenza è gerarchico: le proprietà e i comportamenti di una categoria di oggetti vengono *ereditate* dalle sottocategorie. Le classi `Negoziante`, `Libreria`, `Pasticceria` possono essere graficamente organizzate in una gerarchia come illustrato nella Figura 6.3. In questa gerarchia abbiamo tre classi: la classe `Negoziante`, nella cui interfaccia compare il metodo `pagamentoCC`, e le classi `Pasticceria` e `Libreria`, che *ereditano* dalla

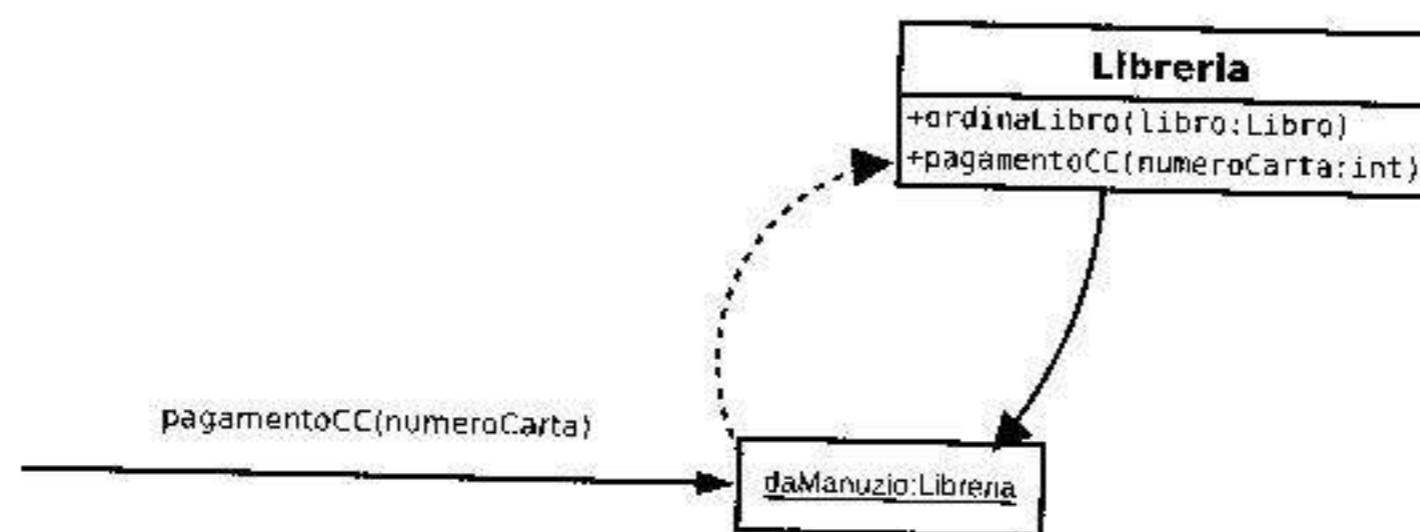


Figura 6.2 Esecuzione del metodo `pagamentoCC`.

classe **Negoziante**. L'ereditarietà è evidenziata dalla freccia che va dalla classe che descrive la categoria più specifica, detta *sottoclasse*, a quella più generale, detta *superclasse*.

In generale, il comportamento e lo stato della superclasse sono ereditati dalla sottoclasse. Nel nostro esempio, il metodo `pagamentoCC`, che fa parte del comportamento della superclasse, è ereditato dalle sottoclassi, e quindi compare implicitamente nell'interfaccia di queste anche se nella rappresentazione grafica non è indicato. Come vedremo meglio quando studieremo l'implementazione della gerarchia, il meccanismo dell'ereditarietà permette di riutilizzare il codice: il fatto che il metodo `pagamentoCC` nella superclasse sia automaticamente disponibile per le sottoclassi significa che non è necessario riscrivere il codice in esse. Ovviamente questo implica che, in fase di esecuzione, quando cercherà il metodo con cui rispondere a un messaggio, la Java Virtual Machine dovrà tener conto della gerarchia.

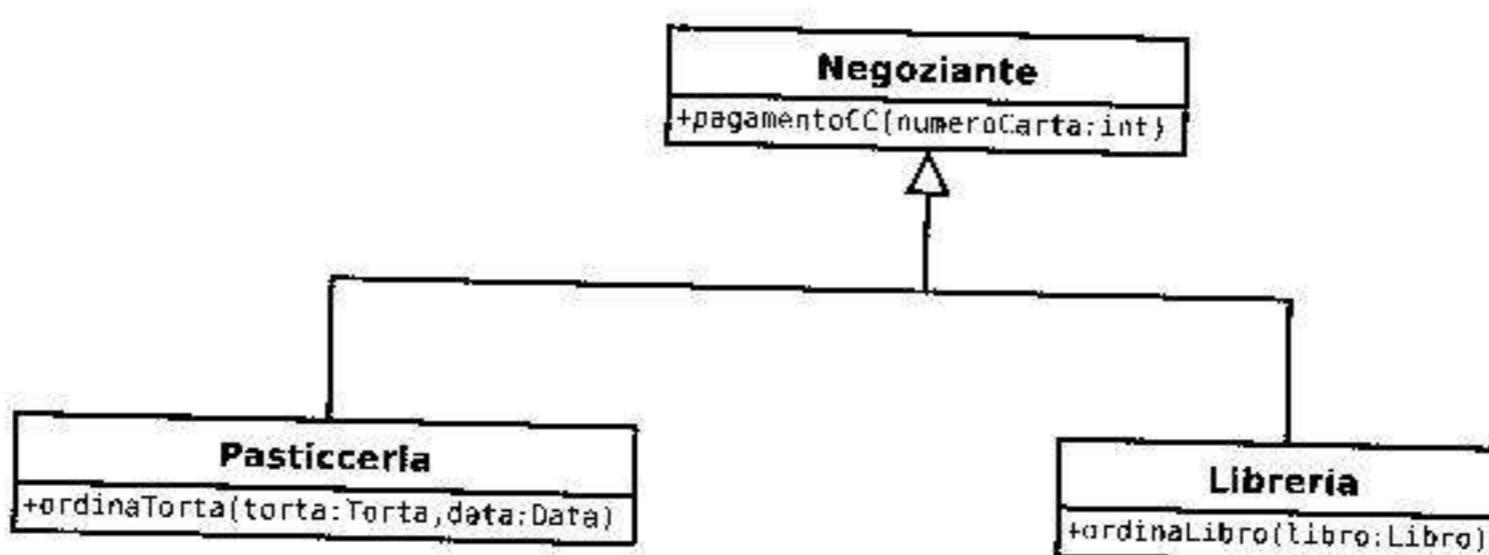


Figura 6.3 Gerarchia **Negoziante**, **Pasticceria**, **Libreria**.

6.1 La classe Rettangolo

In questo paragrafo presentiamo alcuni esempi di uso di una classe **Rettangolo**, che appartiene al package `prog.utili`, impiegata nella presentazione degli argomenti di questo capitolo.

Prima di tutto descriviamo *che cosa fa* la classe. Le sue istanze rappresentano rettangoli. La classe dispone del seguente costruttore:

- `public Rettangolo(double x, double y)`

Costruisce un oggetto che rappresenta un rettangolo, la cui base e la cui altezza hanno le lunghezze fornite, rispettivamente, tramite il primo e il secondo parametro.

La classe dispone inoltre dei seguenti metodi:

- `public double getArea()`

Restituisce l'area del rettangolo rappresentato dall'oggetto che esegue il metodo.

- `public double getPerimetro()`

Restituisce il perimetro del rettangolo rappresentato dall'oggetto che esegue il metodo.

- `public boolean equals(Rettangolo r)`

Confronta il rettangolo rappresentato dall'oggetto che esegue il metodo con il rettangolo di cui viene fornito il riferimento tramite il parametro, restituendo `true` se sono uguali.

- `public boolean haAreaMaggiore(Rettangolo r)`

Restituisce `true` se l'area del rettangolo rappresentato dall'oggetto che esegue il metodo è maggiore di quella del rettangolo di cui viene fornito il riferimento tramite il parametro.

- `public boolean haPerimetroMaggiore(Rettangolo r)`

Restituisce `true` se il perimetro del rettangolo rappresentato dall'oggetto che esegue il metodo è maggiore di quello del rettangolo di cui viene fornito il riferimento tramite il parametro.

- `public String toString()`

Restituisce una stringa di caratteri che descrive il rettangolo rappresentato dall'oggetto che esegue il metodo, come ad esempio "base = 3.1, altezza = 4.3".

- `public double getBase()`

Restituisce la base del rettangolo rappresentato dall'oggetto che esegue il metodo.

- `public double getAltezza()`

Restituisce l'altezza del rettangolo rappresentato dall'oggetto che esegue il metodo.

Come primo esempio d'uso della classe `Rettangolo` scriviamo un'applicazione `ProvaRettangolo` contenente un metodo `main` in cui vengono letti i dati relativi a un rettangolo, e ne vengono stampati i valori dell'area e del perimetro.

Nella prima parte del metodo, dopo avere creato gli oggetti utilizzati per la comunicazione, si chiede all'utente di effettuare l'inserimento dei valori della base e dell'altezza, che vengono assegnati a due variabili, `b` e `a`:

```
ConsoleInputManager in = new ConsoleInputManager();
ConsoleOutputManager out = new ConsoleOutputManager();

out.println("Inserire i dati del rettangolo:");
double b = in.readDouble("base? ");
double a = in.readDouble("altezza? ");
```

È ora possibile costruire l'oggetto che rappresenta il rettangolo richiamando il costruttore all'interno di una espressione `new`. Il risultato dell'espressione, cioè il riferimento all'oggetto costruito, è assegnato a una variabile `r` di tipo `Rettangolo`:

```
Rettangolo r = new Rettangolo(b, a);
```

A questo punto vengono visualizzate le caratteristiche del rettangolo letto. In particolare si utilizza il metodo `toString` per chiedere all'oggetto riferito da `r` di fornire la propria rappresentazione come stringa (come specificato sopra, questa stringa conterrà l'indicazione delle lunghezze della base e dell'altezza); si chiederà inoltre all'oggetto riferito da `r` di eseguire i metodi `getArea` e `getPerimetro` al fine di ottenere i valori dell'area e del perimetro:

```
out.print("Rettangolo letto: ");
out.println(r.toString());
out.println("L'area è " + r.getArea());
out.println("Il perimetro è " + r.getPerimetro());
```

Ecco il testo completo dell'applicazione:

```
import prog.io.*;
import prog.utili.Rettangolo;

class ProvaRettangolo {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //lettura dati
        out.println("Inserire i dati del rettangolo:");
        double b = in.readDouble("base? ");
        double a = in.readDouble("altezza? ");

        //costruzione dell'oggetto
        Rettangolo r = new Rettangolo(b, a);

        //comunicazione del risultato
    }
}
```

```

    out.print("Rettangolo letto: ");
    out.println(r.toString());
    out.println("L'area è " + r.getArea());
    out.println("Il perimetro è " + r.getPerimetro());
}
}

```

Ecco un esempio di esecuzione:

```

Inserire i dati del rettangolo:
base? 5
altezza? 6
Rettangolo letto: base = 5.0, altezza = 6.0
L'area è 30.0
Il perimetro è 22.0

```

Il precedente metodo `main` non effettua alcun controllo sui dati: se uno dei valori inseriti per la base o per l'altezza è negativo, i dati non rappresentano un rettangolo. La descrizione di *che cosa fa* il costruttore della classe `Rettangolo` vista prima non dice nulla in merito a queste situazioni. Quando non si hanno informazioni su come trattare le situazioni anomale, è opportuno fare in modo che esse non si verifichino. Nel caso dell'esempio, prima di richiamare il costruttore della classe `Rettangolo` conviene controllare che la base e l'altezza inserite non siano negative.

Possiamo sostituire le due istruzioni di lettura con cicli che terminano solo quando il dato inserito non è negativo. Ad esempio l'istruzione di lettura della base può essere sostituita da un ciclo della forma:

```

leggi un valore double e assegnaolo alla variabile b
while (il valore letto è negativo) {
    segnala l'errore
    leggi un nuovo valore double e assegnaolo a b
}

```

Tale ciclo può essere codificato come:

```

b = in.readDouble("base? ");
while (b < 0) {
    out.println("Attenzione: la base di un rettangolo " +
               "non può essere negativa!");
    b = in.readDouble("base? ");
}

```

Poiché la condizione del ciclo dipende dal risultato del metodo `readDouble`, è utile scrivere l'operazione di assegnamento *direttamente nella condizione* del ciclo, come segue:

```

while ((b = in.readDouble("base? ")) < 0)
    out.println("Attenzione: la base di un rettangolo " +
               "non può essere negativa!");

```

Ogni volta che viene valutata la condizione del ciclo, si esegue prima di tutto l'assegnamento alla variabile `b` (operazione che richiede a sua volta l'esecuzione del metodo `readDouble`). L'operazione di assegnamento, oltre a modificare la variabile `b`, produce come risultato il valore assegnato, che sarà utilizzato per il confronto con zero nel calcolo della condizione. In sostanza, la valutazione della condizione provoca come *effetto collaterale* una modifica della variabile `b`. Per non rendere i programmi illeggibili, l'uso di effetti collaterali nelle condizioni dev'essere limitato a situazioni semplici come questa.

Ecco il nuovo testo della classe dopo l'inserimento di questi controlli:

```
import prog.io.*;
import prog.utili.Rettangolo;

class ProvaRettangolo {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //lettura dati
        double b, a;
        out.println("Inserire i dati del rettangolo:");
        while ((b = in.readDouble("base? ")) < 0)
            out.println("Attenzione: la base di un rettangolo " +
                        "non può essere negativa!");
        while ((a = in.readDouble("altezza? ")) < 0)
            out.println("Attenzione: l'altezza di un rettangolo " +
                        "non può essere negativa!");

        //costruzione dell'oggetto
        Rettangolo r = new Rettangolo(b, a);

        //comunicazione del risultato
        out.print("Rettangolo letto: ");
        out.println(r.toString());
        out.println("L'area è " + r.getArea());
        out.println("Il perimetro è " + r.getPerimetro());
    }
}
```

A questo punto presentiamo un esempio d'uso della classe `Rettangolo` un po' più complesso del precedente. Scriviamo un'applicazione per determinare il rettangolo con area maggiore in una sequenza di rettangoli fornita da tastiera (se vi sono più rettangoli con la stessa area, maggiore di quella degli altri, l'applicazione comunicherà i dati di uno qualunque di essi).

Il metodo `main` dell'applicazione si basa su due fasi: una di lettura dei dati, l'altra di comunicazione del risultato. La fase di lettura dei dati non è altro che un ciclo in cui, oltre a leggere progressivamente i dati di un rettangolo, si memorizza un riferimento al rettangolo più grande tra quelli letti:

```
do {
    leggi i dati di un rettangolo
    se il rettangolo letto ha area maggiore dei precedenti
        memorizzane il riferimento
} while (l'utente vuole inserire un altro rettangolo)
```

Per la lettura ricorriamo come sempre al metodo `readDouble`, ponendo i valori ottenuti in due variabili di tipo `int`:

```
out.println("Inserisci i dati di un rettangolo:");
x = in.readDouble(" - base? ");
y = in.readDouble(" - altezza? ");
```

Controlliamo dunque i due valori letti; se uno di essi è negativo, segnaliamo l'errore e passiamo direttamente all'iterazione successiva, altrimenti costruiamo l'oggetto memorizzandone il riferimento in una variabile `r`. Subito dopo dovremo confrontare il rettangolo con i precedenti per stabilire se la sua area è maggiore. Tratteremo questo punto in seguito. Come ultima operazione chiediamo all'utente se vuole effettuare un altro inserimento. A questo proposito domandiamo all'oggetto `in` di eseguire il metodo `readSiNo` che richiede all'utente di inserire uno dei due caratteri, `s` o `n`, e restituisce `true` nel caso sia stato inserito `s`. Ecco la struttura del ciclo che abbiamo delineato:

```
Rettangolo r;
boolean continuare;
double x, y;

do {
    //legge i dati di un rettangolo
    out.println("Inserisci i dati di un rettangolo:");
    x = in.readDouble(" - base? ");
    y = in.readDouble(" - altezza? ");

    //controlla i dati e, se sono corretti, costruisce l'oggetto
    if (x < 0 || y < 0)
        out.println("I dati inseriti non rappresentano un rettangolo");
    else {
        r = new Rettangolo(x, y);
        out.println("Rettangolo: ");
        out.println(" " + r.toString());
        out.println(" area = " + r.getArea() +
```

```

    ", perimetro = " + r.getPerimetro());
    ...
    ...confronto per il calcolo del rettangolo con area max...
}
out.println();

//chiede all'utente se intende effettuare un altro inserimento
continuare = in.readSiNo("Vuoi inserire i dati di un altro " +
    "rettangolo? (s/n) ");
} while (continuare);

```

Per individuare il rettangolo con area maggiore introduciamo una variabile `rAreaMax` il cui scopo è di riferirsi, di volta in volta, al rettangolo più grande tra quelli letti. Prima del ciclo la variabile è inizializzata a `null`. All'interno del ciclo confrontiamo il rettangolo riferito da `r` con quello riferito da `rAreaMax`. In particolare, per controllare se il rettangolo riferito da `r` abbia area maggiore di quello riferito da `rMax`, chiediamo al rettangolo riferito da `r` di eseguire il proprio metodo `haAreaMaggior`e fornendo come argomento `rAreaMax`. Se il risultato ottenuto è `true`, modifichiamo il riferimento `rAreaMax`:

```

if (r.haAreaMaggior(rAreaMax))
    rAreaMax = r;

```

Il precedente confronto ha senso quando `rAreaMax` si riferisce effettivamente a un rettangolo. Tuttavia, quando il confronto viene effettuato per la prima volta, cioè dopo avere letto il primo rettangolo, `rAreaMax` contiene `null`. In questo caso bisogna assegnare `r` direttamente a `rAreaMax`, senza controllare la condizione. Possiamo pertanto riscrivere la selezione così:

```

if (rAreaMax == null || r.haAreaMaggior(rAreaMax))
    rAreaMax = r;

```

Si osservi che, grazie alla *lazy evalutation*, la chiamata `r.haAreaMaggior(rAreaMax)` è effettuata solo quando `rAreaMax` non è `null`; se così non fosse, si avrebbe un errore in fase di esecuzione. All'uscita dal ciclo viene infine comunicato il risultato.

Ecco il testo completo dell'applicazione:

```

import prog.io.*;
import prog.utili.Rettangolo;

class RettangoloAreaMax {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

```

```
Rettangolo r, rAreaMax = null;
boolean continuare;
double x, y;

do {
    //legge i dati di un rettangolo
    out.println("Inserisci i dati di un rettangolo:");
    x = in.readDouble(" - base? ");
    y = in.readDouble(" - altezza? ");

    //controlla i dati e, se sono corretti, costruisce l'oggetto
    if (x < 0 || y < 0)
        out.println("I dati inseriti non rappresentano un rettangolo");
    else {
        r = new Rettangolo(x, y);
        out.println("Rettangolo: ");
        out.println(" " + r.toString());
        out.println(" area = " + r.getArea() +
                   ", perimetro = " + r.getPerimetro());

        //confronta il rettangolo con quello di area maggiore
        if (rAreaMax == null || r.haAreaMaggiorer(rAreaMax))
            rAreaMax = r;
    }
    out.println();

    //chiede all'utente se intende effettuare un altro inserimento
    continuare = in.readSiNo("Vuoi inserire i dati di un altro " +
                             "rettangolo? (s/n) ");
} while (continuare);

//comunica le caratteristiche del rettangolo di area maggiore
if (rAreaMax == null)
    out.println("Non è stato inserito alcun rettangolo");
else {
    out.println("Rettangolo di area maggiore: ");
    out.println(" " + rAreaMax.toString());
    out.println(" area = " + rAreaMax.getArea() +
               ", perimetro = " + rAreaMax.getPerimetro());
}
```

Ecco un primo esempio di esecuzione, in cui vengono inseriti i dati di tre rettangoli:

Inserisci i dati di un rettangolo:

- base? 7
- altezza? 3

Rettangolo:

base = 7.0, altezza = 3.0
area = 21.0, perimetro = 20.0

Vuoi inserire i dati di un altro rettangolo? (s/n) s

Inserisci i dati di un rettangolo:

- base? 2.5
- altezza? 6

Rettangolo:

base = 2.5, altezza = 6.0
area = 15.0, perimetro = 17.0

Vuoi inserire i dati di un altro rettangolo? (s/n) s

Inserisci i dati di un rettangolo:

- base? 4
- altezza? 2.2

Rettangolo:

base = 4.0, altezza = 2.2
area = 8.8, perimetro = 12.4

Vuoi inserire i dati di un altro rettangolo? (s/n) n

Rettangolo di area maggiore:

base = 7.0, altezza = 3.0
area = 21.0, perimetro = 20.0

Ecco un secondo esempio in cui l'utente inserisce dati scorretti e sceglie poi di terminare l'esecuzione:

Inserisci i dati di un rettangolo:

- base? 7
- altezza? -3

I dati inseriti non rappresentano un rettangolo

Vuoi inserire i dati di un altro rettangolo? (s/n) n

Non è stato inserito alcun rettangolo

6.2 UML: rappresentazione di classi e oggetti

In questo paragrafo introdurremo i primi elementi di UML (*Unified modeling language*), un linguaggio grafico per la modellazione di sistemi a oggetti. UML è utilizzato per la progettazione e l'analisi di sistemi software; noi utilizzeremo alcune notazioni grafiche per chiarire certi aspetti del linguaggio Java. È importante notare che UML non è stato sviluppato specificamente per la progettazione di programmi Java, ma per la progettazione di sistemi software basati su un generico linguaggio a oggetti. Questa è la ragione per cui la sintassi di UML non coincide con quella di Java. Inoltre, mentre tutti i concetti di Java hanno un corrispondente in UML, quest'ultimo permette di modellare anche concetti non previsti da Java.

Dato che l'idea di classe ha un ruolo cruciale nell'ambito della programmazione a oggetti, UML prevede una notazione per la sua rappresentazione. In UML una classe è rappresentata graficamente da un rettangolo diviso in tre compatti:

- nel comparto più in alto è indicato il nome della classe;
- nel comparto di mezzo sono indicati i campi della classe (che studieremo in dettaglio più avanti nella parte relativa all'implementazione delle classi);
- nel comparto più in basso sono indicati i costruttori e le operazioni che la classe mette a disposizione.

A seconda del grado di dettaglio con cui si vuole rappresentare una classe è possibile evidenziare o nascondere le informazioni relative ai campi e alle operazioni. Nella Figura 6.4 sono proposte, a titolo d'esempio, due diverse rappresentazioni della classe *Rettangolo*. Nella prima è evidenziato solo il nome della classe, mentre nella seconda sono evidenziati tutti e tre i compatti, anche se quello dei campi è vuoto.

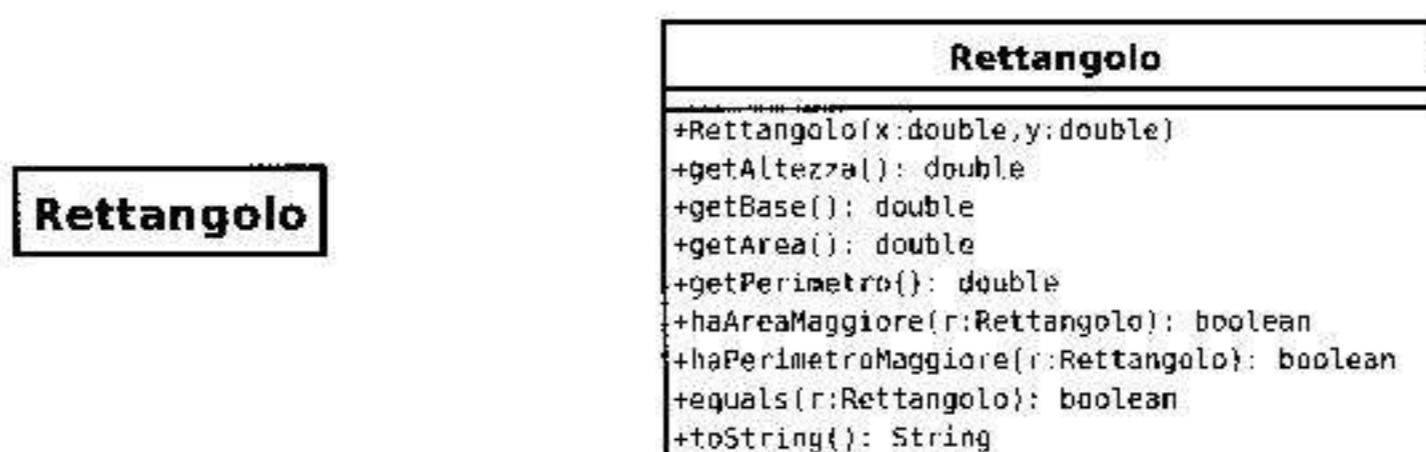


Figura 6.4 UML: visualizzazioni diverse della classe *Rettangolo*.

Osserviamo che la sintassi di UML per la descrizione delle operazioni è leggermente diversa da quella di Java; per la precisione la sintassi è la seguente:

nome_operazione (*lista_dei_parametri*) : *tipo_restituito*

Quindi, diversamente da Java, dove il tipo restituito è indicato prima della segnatura del metodo, in UML il tipo è indicato dopo la segnatura. Per quel che riguarda i parametri, il discorso è analogo; la sintassi di dichiarazione di un parametro è:

nome_parametro : tipo

mentre la lista dei parametri è una sequenza di dichiarazioni di parametro separate da virgole. Il simbolo + che precede la dichiarazione dell'operazione indica che la visibilità dell'operazione è pubblica: è pertanto il corrispondente della parola chiave `public` di Java.

Per quel che riguarda gli oggetti, anch'essi sono rappresentati graficamente tramite rettangoli; all'interno del rettangolo che rappresenta l'oggetto viene inserito il nome dell'istanza (la variabile riferimento che ne tiene traccia) seguito dal carattere due punti e dal nome della classe; il tutto viene sottolineato. La sintassi è:

nomeIstanza : nome_classe

I messaggi sono rappresentati mediante una freccia diretta verso l'oggetto cui è inviato il messaggio e da un'etichetta che specifica il messaggio. Ad esempio, nella Figura 6.5 è rappresentato un oggetto *r* di tipo *Rettangolo* che riceve il messaggio `toString()`.

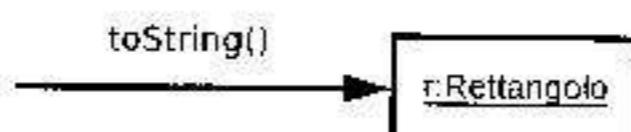


Figura 6.5 UML: oggetti e messaggi.

6.3 La classe Quadrato

Illustriamo ora brevemente un'altra classe del package `prog.utili` che, dal punto di vista logico, è imparentata con la classe *Rettangolo*: la classe *Quadrato*. Mostreremo poi che il legame che intercorre dal punto di vista logico tra le due classi (derivante dai legami tra i tipi di oggetti rappresentati) può essere esplicitato nella definizione delle classi tramite il meccanismo dell'*ereditarietà*.

Anche in questo caso cominciamo descrivendo *che cosa fa* la classe le cui istanze rappresentano quadrati.

La classe dispone del seguente costruttore:

- `public Quadrato(double x)`

Costruisce un oggetto che rappresenta un quadrato il cui lato ha la lunghezza fornita tramite il parametro.

La classe dispone inoltre dei seguenti metodi (molti dei quali analoghi a quelli di *Rettangolo*):

- `public double getArea()`

Restituisce l'area del quadrato rappresentato dall'oggetto che esegue il metodo.

- `public double getPerimetro()`

Restituisce il perimetro del quadrato rappresentato dall'oggetto che esegue il metodo.

- **public boolean equals(Quadrato q)**
Confronta il quadrato rappresentato dall'oggetto che esegue il metodo con il quadrato di cui viene fornito il riferimento tramite il parametro, restituendo true se sono uguali.
- **public boolean haAreaMaggiore(Quadrato r)**
Restituisce true se l'area del quadrato rappresentato dall'oggetto che esegue il metodo è maggiore di quella del quadrato di cui viene fornito il riferimento tramite il parametro.
- **public boolean haPerimetroMaggiore(Quadrato r)**
Restituisce true se il perimetro del quadrato rappresentato dall'oggetto che esegue il metodo è maggiore di quello del quadrato di cui viene fornito il riferimento tramite il parametro.
- **public String toString()**
Restituisce una stringa di caratteri che descrive il quadrato rappresentato dall'oggetto, come ad esempio "lato = 3.1".
- **public double getLato()**
Restituisce il lato del rettangolo rappresentato dall'oggetto.

Ora possiamo scrivere applicazioni che utilizzano la classe Quadrato. Come abbiamo fatto nel caso dei rettangoli, possiamo ad esempio scrivere un'applicazione che legge i lati di un quadrato e ne visualizza l'area e il perimetro:

```
import prog.io.*;
import prog.utili.Quadrato;

class ProvaQuadrato {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //lettura dati
        double lato;
        while ((lato = in.readDouble("Quanto è lungo il lato " +
            "del quadrato? ")) < 0)
            out.println("Attenzione: il lato di un quadrato non " +
                "può essere negativo!");

        //costruzione dell'oggetto
        Quadrato q = new Quadrato(lato);

        //comunicazione del risultato
        out.println("L'area del quadrato è " + q.getArea());
        out.println("Il perimetro del quadrato è " + q.getPerimetro());
    }
}
```

```

    out.print("Quadrato letto: ");
    out.println(q.toString());
    out.println("L'area è " + q.getArea());
    out.println("Il perimetro è " + q.getPerimetro());
}
}

```

Potremmo anche scrivere un'applicazione che legga i dati relativi a una sequenza di quadrati e determini quello con area maggiore rispetto agli altri. Dal momento che determinare il quadrato con area maggiore equivale però a determinare il quadrato con lato maggiore, questo esempio, oltre a essere del tutto simile a quello presentato nel caso dei rettangoli, non è molto significativo.

È molto più interessante sviluppare applicazioni in grado di trattare sia rettangoli sia quadrati. In particolare, partiamo dalla classe RettangoloAreaMax per sviluppare un'applicazione FiguraAreaMax in grado di ricevere i dati relativi a una sequenza contenente rettangoli e quadrati, e determinare quale tra essi abbia area maggiore. Riportiamo qui il codice del metodo main di RettangoloAreaMax: conserviamo le parti che possono essere mantenute o leggermente modificate ed evidenziamo quelle da modificare:

```

public static void main(String[] args) {
    //predisposizione dei canali di comunicazione
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();

    ...tipi?... r, rAreaMax = null;
    boolean continuare;
    ...altre variabili utili per la lettura...

    do {
        //legge i dati di una figura
        ...fase di lettura...;

        //controlla i dati e, se sono corretti, costruisce l'oggetto
        if (...dati non sono corretti...)
            out.println(...messaggio d'errore...);
        else {
            ...costruisci l'oggetto che rappresenta la figura e
            memorizzane il riferimento in r...
            ...visualizza le caratteristiche della figura appena
            letta...

        //confronta la figura con quella di area maggiore
        if (rAreaMax == null || r.haAreaMaggior(rAreaMax))
            rAreaMax = r;
    }
}

```

```

    }
    out.println();

    //chiede all'utente se intenda effettuare un altro inserimento
    continuare = in.readSiNo("Vuoi inserire i dati di un'altra " +
        "figura? (s/n) ");
} while (continuare);

//comunica le caratteristiche della figura di area maggiore
if (rAreaMax == null)
    out.println("Non è stata inserita alcuna figura");
else {
    out.println("Figura di area maggiore: ");
    out.println(" " + rAreaMax.toString());
    out.println(" area = " + rAreaMax.getArea() +
        ", perimetro = " + rAreaMax.getPerimetro());
}
}
}

```

È importante osservare quanto segue.

- Quando costruiamo l'oggetto corrispondente alla figura letta, memorizziamo il riferimento a esso in una variabile di nome `r`. Poiché non conosciamo a priori il tipo di figura letta, dovremo poter utilizzare il riferimento `r` sia per i quadrati sia per i rettangoli.
- Analogamente, la variabile `rAreaMax` mantiene il riferimento alla figura di area maggiore tra quelle lette. In alcuni momenti dell'esecuzione essa potrà essere un rettangolo, in altri un quadrato. Pertanto anche il tipo del riferimento `rAreaMax` dev'essere valido sia per i rettangoli sia per i quadrati.
- Per confrontare le figure in base alle aree utilizziamo il metodo `haAreaMaggiore`: questo metodo dovrà essere in grado di confrontare non solo due quadrati o due rettangoli, ma anche un quadrato e un rettangolo.
- Alla fine, per comunicare il risultato viene chiamato, tra gli altri, il metodo `toString` dell'oggetto riferito da `rAreaMax`. Ci aspettiamo che questo metodo comunichi i valori della base e dell'altezza qualora la figura più grande sia un rettangolo, e comunichi il valore del lato qualora la figura più grande sia un quadrato. In altre parole, desideriamo che il metodo `toString` effettivamente eseguito dipenda dal tipo di figura riferita da `rAreaMax`.

Vedremo ora come realizzare i punti precedenti. In particolare, grazie alla *gerarchia delle classi* e al meccanismo dell'*ereditarietà*, potremo utilizzare un unico tipo riferimento (in questo caso di tipo `Rettangolo`) per riferirci, tramite le variabili `r` e `rAreaMax`, sia a quadrati sia a rettangoli, mentre grazie al *polimorfismo* potremo far sì che una stessa invocazione, come `rAreaMax.toString()`, selezioni il metodo appropriato per l'oggetto effettivo (quadrato o rettangolo) cui è rivolta.

6.4 Ereditarietà e polimorfismo

Un quadrato è un particolare tipo di rettangolo in cui la base e l'altezza hanno la stessa lunghezza. L'insieme dei quadrati è pertanto un sottoinsieme dell'insieme dei rettangoli. Questo legame può essere evidenziato definendo la classe Quadrato come *sottoclasse* della classe Rettangolo. Per farlo, chi definisce la classe Quadrato dovrà dichiarare nella sua intestazione che essa *estende* Rettangolo:

```
class Quadrato extends Rettangolo
```

In questo modo la classe Quadrato *eredita* i metodi e i campi definiti nella classe Rettangolo. La classe Rettangolo è anche detta *superclasse* di Quadrato.

In UML la relazione esistente fra la classe Rettangolo e la classe Quadrato viene modellata dalla *relazione di generalizzazione*, rappresentata tramite una freccia a punta triangolare come nella Figura 6.6. La direzione della freccia indica che ogni Quadrato è un caso particolare di un

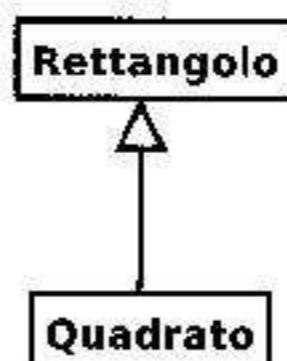


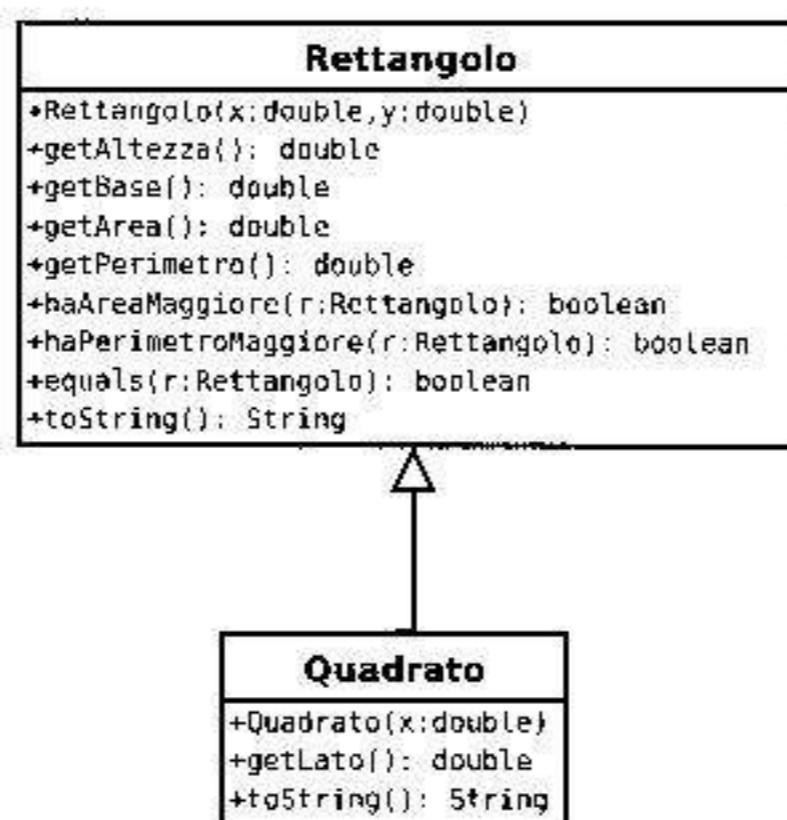
Figura 6.6 UML: rappresentazione dell'ereditarietà.

Rettangolo o (per render conto del nome della relazione in UML) che la classe Rettangolo modella un concetto più generale di quello modellato dalla classe Quadrato.

Grazie al meccanismo dell'*ereditarietà*, chi implementa la classe Quadrato dovrà limitarsi a indicare solo quanto si differenzia da Rettangolo. In base alla descrizione che abbiamo dato delle due classi, i metodi `toString` dovranno ad esempio essere differenti; inoltre Quadrato avrà il metodo `getLato`, che non è presente in Rettangolo. Per il calcolo dell'area e del perimetro, la classe Quadrato potrebbe invece utilizzare gli stessi metodi di Rettangolo. In UML queste scelte vengono rappresentate indicando nella sottoclassse solo i metodi per i quali questa si distingue dalla superclasse. Ad esempio nella Figura 6.7 è evidenziato che la classe Quadrato si differenzia dalla classe Rettangolo per il metodo `toString`, che compare in ambedue le classi, e per il metodo `getLato`, che compare solo nella classe Quadrato. Tutte le operazioni definite in Rettangolo, a parte `toString`, sono invece a disposizione della classe Quadrato per il semplice fatto che vengono ereditate.

Un primo scopo dell'ereditarietà è dunque quello di poter realizzare varianti di classi esistenti, riutilizzandone l'implementazione senza dover riscrivere tutto da zero. Studieremo questi aspetti nei capitoli successivi, quando esamineremo l'implementazione delle classi.

Sebbene la possibilità di riutilizzare il codice possa sembrare in un primo momento il fine e il significato fondamentale dell'ereditarietà, essa ne è solo una caratteristica. La vera potenza dell'ereditarietà è legata alle possibilità che offre in relazione all'uso dei tipi riferimento.

**Figura 6.7** UML: metodi ereditati e ridefiniti.

Studiando i tipi primitivi abbiamo visto che un valore di un tipo può essere sempre assegnato, mediante una promozione, a una variabile di un tipo più ampio. Ad esempio è possibile assegnare un valore di tipo `int` a una variabile di tipo `long`.

Nel caso dei riferimenti agli oggetti esiste un meccanismo analogo. In particolare il riferimento a un oggetto di una sottoclasse può essere sempre assegnato a una variabile il cui tipo sia una superclasse della classe alla quale appartiene l'oggetto. Diciamo anche che il nome della superclasse è un *supertipo* per il tipo della sottoclasse.

Nell'esempio che stiamo sviluppando il riferimento a un oggetto della classe `Quadrato` (sottoclasse) può essere assegnato a una variabile dichiarata di tipo `Rettangolo` (superclasse): poiché un quadrato è un rettangolo (così come un valore `int` è anche un valore `long`), una variabile di tipo `Rettangolo` può riferirsi a un oggetto di tipo `Quadrato` (così come una variabile di tipo `long` può contenere anche valori del tipo `int`). Dunque `Rettangolo` è un supertipo di `Quadrato`: i riferimenti a oggetti della classe `Quadrato` possono essere promossi al tipo `Rettangolo`.

In maniera analoga, quando si invoca un metodo che si aspetta un parametro di un tipo riferimento, l'argomento fornito può essere di un qualunque sottotipo. Ad esempio, supponendo di disporre di un metodo `m` con un parametro di tipo `Rettangolo` e supponendo che `q` sia una variabile di tipo `Quadrato`, è possibile invocare `m` fornendo `q` come argomento. Al momento dell'invocazione, il riferimento presente in `q` sarà promosso al tipo più ampio.

Consideriamo ora le seguenti linee di codice:

```

Rettangolo r;
Quadrato q = new Quadrato(6);
r = q;

```

Avremmo potuto anche scrivere direttamente:

```

Rettangolo r = new Quadrato(6);

```

Supponiamo ora di chiedere all'oggetto riferito da `r` di eseguire il proprio metodo `toString()`, mediante la chiamata

```
r.toString();
```

Ricordiamo che ci sono due metodi `toString`: uno per la classe `Rettangolo` e uno per la classe `Quadrato`. Infatti il metodo `toString` è stato *ridefinito* o *riscritto* nella classe `Quadrato`. Quale dei due metodi sarà effettivamente invocato *viene determinato in fase d'esecuzione* in base alla classe cui appartiene l'oggetto costruito, e *non* in base al tipo del riferimento. Parliamo in questo caso di *polimorfismo* (più forme): in momenti diversi una stessa chiamata, come `r.toString()`, può invocare metodi differenti sulla base della classe cui appartiene l'oggetto al quale viene chiesto di eseguire il metodo:

- ogni oggetto conosce la classe cui appartiene (determinata in base al costruttore che l'ha creato);
- quando si chiede a un oggetto di eseguire un metodo, questo viene cercato anzitutto tra i metodi della classe cui appartiene l'oggetto e, solo nel caso non sia presente tra essi, tra i metodi della superclasse.

In questo caso l'oggetto riferito da `r` è stato creato con il costruttore `Quadrato`, e dunque è un'istanza della classe `Quadrato`. Pertanto la chiamata `r.toString()` provoca l'esecuzione del metodo `toString` della classe `Quadrato`, nonostante `r` sia un riferimento di tipo `Rettangolo`. Riassumendo, possiamo affermare che *il metodo effettivamente eseguito è stabilito in esecuzione in base all'oggetto e non al riferimento*.

La stringa ottenuta dalla precedente chiamata è quindi:

```
lato = 6
```

sebbene il riferimento `r` sia di tipo `Rettangolo`.

È possibile scrivere una condizione per controllare il tipo di un oggetto utilizzando l'operatore `instanceof`. Quest'operatore binario ha come operando sinistro un riferimento e come operando destro un tipo riferimento. Per ora gli unici tipi riferimento che conosciamo sono i nomi delle classi. Se l'oggetto associato al riferimento indicato a sinistra è un'istanza della classe indicata a destra dell'operatore, il risultato dell'operatore è `true`.

Supponiamo, ad esempio, di avere dichiarato:

```
Rettangolo r;
```

L'espressione

```
r instanceof Rettangolo
```

è vera se l'oggetto riferito da `r` è un'istanza della classe `Rettangolo`. In questo caso:

- se `r` contiene `null`, il risultato dell'espressione è `false`, perché non vi è alcun oggetto associato a `r`;

- se `r` è diverso da `null`, il risultato è `true`, perché in base al tipo di `r` l'oggetto associato dev'essere per forza un'istanza di `Rettangolo` (è fondamentale ricordare che ogni quadrato è anche un rettangolo, cioè ogni istanza di `Quadrato` è una particolare istanza di `Rettangolo`).

Consideriamo ora l'espressione:

```
r instanceof Quadrato
```

Per definizione essa è vera solo se `r` si riferisce a un oggetto della classe `Quadrato`. Dunque essa è falsa nel caso in cui `r` contenga `null` e nel caso in cui `r` si riferisca a un'istanza di `Rettangolo` che non sia anche un'istanza di `Quadrato`, cioè qualora l'oggetto riferito da `r` non sia stato costruito utilizzando il costruttore di `Quadrato`.

Utilizzando quanto abbiamo appena descritto, possiamo ora scrivere il codice dell'applicazione `FiguraAreaMax`. Abbiamo osservato che i riferimenti `r` e `rAreaMax` devono potersi riferire sia a istanze di `Quadrato` sia a istanze di `Rettangolo`. A tale scopo è sufficiente dichiararli di tipo `Rettangolo`.

Prima di scrivere la codifica dell'intero metodo `main` esaminiamone la parte finale in cui viene comunicato il risultato, che è già stata codificata come segue:

```
if (rAreaMax == null)
    out.println("Non è stata inserita alcuna figura");
else {
    out.println("Figura di area maggiore: ");
    out.println(" " + rAreaMax.toString());
    out.println(" area = " + rAreaMax.getArea() +
               ", perimetro = " + rAreaMax.getPerimetro());
}
```

Grazie al polimorfismo, il metodo `toString` selezionato nella quinta riga del codice riportato dipenderà dal tipo effettivo dell'oggetto riferito da `rAreaMax`. In altre parole, se l'oggetto è un'istanza di `Quadrato`, sarà eseguito il metodo `toString` di `Quadrato`, altrimenti quello di `Rettangolo`. Possiamo migliorare la comunicazione del risultato in modo che in luogo della riga:

Figura di area maggiore:

compaia in output il messaggio:

La figura di area maggiore è un quadrato:

oppure:

La figura di area maggiore è un rettangolo:

Per decidere se scrivere la parola `quadrato` oppure la parola `rettangolo` esaminiamo il tipo dell'oggetto riferito da `rAreaMax` utilizzando l'operatore `instanceof`.

Dopo avere introdotto questa modifica, la parte di codice riportata in precedenza può essere scritta come segue:

```
if (rAreaMax == null)
    out.println("Non è stata inserita alcuna figura");
else {
    out.print("La figura di area maggiore è un ");
    if (rAreaMax instanceof Quadrato)
        out.println("quadrato: ");
    else
        out.println("rettangolo: ");
    out.println(" " + rAreaMax.toString());
    out.println(" area = " + rAreaMax.getArea() +
               ", perimetro = " + rAreaMax.getPerimetro());
}
```

Presentiamo ora la codifica dell'intera applicazione. All'interno del ciclo del metodo `main` vengono letti i dati (base e altezza) di una figura. Se i valori inseriti coincidono, il metodo invoca il costruttore di `Quadrato` fornendo come argomento il valore della base, altrimenti invoca quello di `Rettangolo` fornendo i due valori letti. In ogni caso, il riferimento all'oggetto così costruito è assegnato alla variabile `r`.

```
import prog.io.*;
import prog.utili.Figura;
import prog.utili.Rettangolo;
import prog.utili.Quadrato;

class FiguraAreaMax {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        Rettangolo r, rAreaMax = null;
        boolean continuare;
        double x, y;

        do {
            //legge i dati della figura
            out.println("Inserisci i dati della figura:");
            x = in.readDouble("- base? ");
            y = in.readDouble("- altezza? ");
            if (x == y)
                rAreaMax = new Quadrato(x);
            else
                rAreaMax = new Rettangolo(x, y);
            continuare = in.readBoolean("Continua? ");
        } while (continuare);
    }
}
```

```
//controlla i dati e, se sono corretti, costruisce l'oggetto
if (x < 0 || y < 0)
    out.println("I dati inseriti non rappresentano né un " +
                "quadrato, né un rettangolo");
else {
    if (x == y) {
        r = new Quadrato(x);
        out.println("La figura è un quadrato:");
    } else {
        r = new Rettangolo(x, y);
        out.println("La figura è un rettangolo:");
    }
    out.println(" " + r.toString());
    out.println(" area = " + r.getArea() +
               ", perimetro = " + r.getPerimetro());
}

//confronta la figura con quella di area maggiore
if (rAreaMax == null || r.haAreaMaggiore(rAreaMax))
    rAreaMax = r;
}
out.println();

//chiede all'utente se intenda effettuare un altro inserimento
continuare = in.readSiNo("Vuoi inserire i dati di un'altra " +
                         "figura? (s/n) ");
} while (continuare);

//comunica le caratteristiche della figura di area maggiore
if (rAreaMax == null)
    out.println("Non è stata inserita alcuna figura");
else {
    out.print("La figura di area maggiore è un ");
    if (rAreaMax instanceof Quadrato)
        out.println("quadrato: ");
    else
        out.println("rettangolo: ");
    out.println(" " + rAreaMax.toString());
    out.println(" area = " + rAreaMax.getArea() +
               ", perimetro = " + rAreaMax.getPerimetro());
}
}
```

Ecco un esempio d'esecuzione:

Inserisci i dati della figura:

- base? 7
- altezza? 4

La figura è un rettangolo:

base = 7.0, altezza = 4.0
area = 28.0, perimetro = 22.0

Vuoi inserire i dati di un'altra figura? (s/n) s

Inserisci i dati della figura:

- base? 6
- altezza? 6

La figura è un quadrato:

lato = 6.0
area = 36.0, perimetro = 24.0

Vuoi inserire i dati di un'altra figura? (s/n) s

Inserisci i dati della figura:

- base? 4
- altezza? 5

La figura è un rettangolo:

base = 4.0, altezza = 5.0
area = 20.0, perimetro = 18.0

Vuoi inserire i dati di un'altra figura? (s/n) n

La figura di area maggiore è un quadrato:

lato = 6.0
area = 36.0, perimetro = 24.0

Nell'analisi precedente abbiamo affermato che ognuna delle due classi, Quadrato e Rettangolo, ha un proprio metodo `haAreaMaggiori`. Il metodo presente in Rettangolo (che riceve un argomento di tipo Rettangolo) sarebbe già sufficiente per confrontare sia quadrati sia rettangoli. In realtà, come preciseremo nel paragrafo successivo, ciò che ci interessa utilizzare è un metodo per il confronto delle aree di due figure geometriche qualsiasi.

6.5 Le classi astratte

Riprendiamo l'ultimo esempio che abbiamo visto, cioè l'applicazione `FiguraAreaMax`. Perché dovremmo limitarci a considerare solo quadrati e rettangoli? Non possiamo scrivere l'applicazione `FiguraAreaMax` in modo che possa confrontare anche altri tipi di figure geometriche, ad esempio cerchi e triangoli?

Per semplicità ci limitiamo a considerare, oltre a quadrati e rettangoli, anche i cerchi; la trattazione può essere facilmente estesa ad altre figure.

Supponiamo dunque di disporre di una classe `Cerchio`, con metodi analoghi a quelli di `Quadrato` e di `Rettangolo`. Più precisamente, supponiamo che la classe fornisca il seguente costruttore:

- `public Cerchio(double r)`

Costruisce un oggetto che rappresenta il cerchio il cui raggio è specificato dall'argomento.

Supponiamo inoltre che la classe disponga dei seguenti metodi:

- `public double getArea()`

Restituisce l'area del cerchio rappresentato dall'oggetto che esegue il metodo.

- `public double getCirconferenza()`

`public double getPerimetro()`

Entrambi restituiscono la lunghezza della circonferenza dell'oggetto che esegue il metodo (supponiamo che vi sia anche il metodo `getPerimetro` per uniformità con le altre figure considerate).

- `public boolean equals(Cerchio c)`

Confronta il cerchio rappresentato dall'oggetto che esegue il metodo con il cerchio di cui è fornito il riferimento tramite il parametro, restituendo `true` se sono uguali.

- `public boolean haAreaMaggiore(Cerchio c)`

Restituisce `true` se l'area del cerchio rappresentato dall'oggetto che esegue il metodo è maggiore di quella del cerchio di cui è fornito il riferimento tramite il parametro.

- `public boolean haPerimetroMaggiore(Cerchio c)`

Restituisce `true` se il perimetro del cerchio rappresentato dall'oggetto che esegue il metodo è maggiore di quello del cerchio di cui è fornito il riferimento tramite il parametro.

- `public String toString()`

Restituisce una stringa di caratteri che descrive il cerchio rappresentato dall'oggetto, come ad esempio "raggio = 3.1".

- `public double getRaggio()`

Restituisce il raggio del cerchio rappresentato dall'oggetto che esegue il metodo.

Iniziamo ora a descrivere la struttura del metodo `main` della nuova versione dell'applicazione `FiguraAreaMax`.

L'idea è quella di proporre all'utente un menu con il quale selezionare il tipo di figura da inserire tra quelle disponibili. In base alla scelta fatta, vengono richiesti i dati necessari per definire la figura. Come nella versione precedente, questa fase è all'interno di un ciclo in cui l'utente può inserire una nuova figura a ogni iterazione. A ogni inserimento si effettua un confronto tra la figura appena inserita e quella, tra le precedenti, che aveva area maggiore.

In altre parole possiamo seguire lo schema della versione precedente:

```

... r, rAreaMax = null;
boolean continuare;
...

do {
    //legge i dati di una figura
    ...
    //controlla i dati e, se sono corretti, costruisce l'oggetto
    ...
    //confronta la figura con quella di area maggiore
    if (rAreaMax == null || r.haAreaMaggiori(rAreaMax))
        rAreaMax = r;
    ...
} while (continuare);

```

Per i riferimenti `r` e `rAreaMax` è stato utilizzato, nella versione precedente, il tipo `Rettangolo`, che è adeguato anche per istanze di `Quadrato`, perché quest'ultima classe è una sottoclasse di `Rettangolo`. Nel caso dei cerchi, invece, non esiste alcuna relazione di sottoclasse o superclasse con quadrati e rettangoli. Dunque non è possibile utilizzare immediatamente lo stesso meccanismo o, meglio, per utilizzarci lo stesso meccanismo dovremmo riuscire a definire `r` e `rAreaMax` di un tipo adatto per `Rettangolo`, `Quadrato` e `Cerchio`, cioè di un supertipo dei tre tipi.

Così come abbiamo osservato che i quadrati sono rettangoli particolari, possiamo osservare che rettangoli, quadrati e cerchi sono particolari figure geometriche. Le tre classi, `Rettangolo`, `Quadrato` e `Cerchio`, sono state progettate tenendo conto di questo. In particolare esse sono sottoclassi di una classe più ampia denominata `Figura`, con cui si intende appunto rappresentare tutte le figure geometriche nel piano.

Le classi `Rettangolo` e `Cerchio` sono state definite estendendo direttamente `Figura`. Pertanto sono sue sottoclassi dirette:

```

public class Rettangolo extends Figura
public class Cerchio extends Figura

```

D'altra parte la classe `Quadrato` è una sottoclasse di `Rettangolo`, e quindi (indirettamente) anche di `Figura`. Le relazioni appena indicate definiscono la gerarchia tra le quattro classi che stiamo considerando (Figura 6.8). Così come `Rettangolo` è un supertipo per `Quadrato`, e pertanto riferimenti di tipo `Rettangolo` possono essere utilizzati anche per riferirsi a istanze di `Quadrato`, il tipo `Figura` è un supertipo comune a tutte le classi riportate. Perciò potremo utilizzare riferimenti dichiarati di tipo `Figura` per riferirci a istanze di `Rettangolo`, di `Quadrato` e di `Cerchio` (e di tutte le altre sottoclassi che possono trovarsi, o che potranno essere aggiunte, sotto `Figura` nella gerarchia).

Riportiamo ora il codice della nuova versione di `FiguraAreaMax`, progettato secondo quanto descritto sopra. In particolare poniamo l'accento sul fatto che le variabili `r` e `rAreaMax` sono

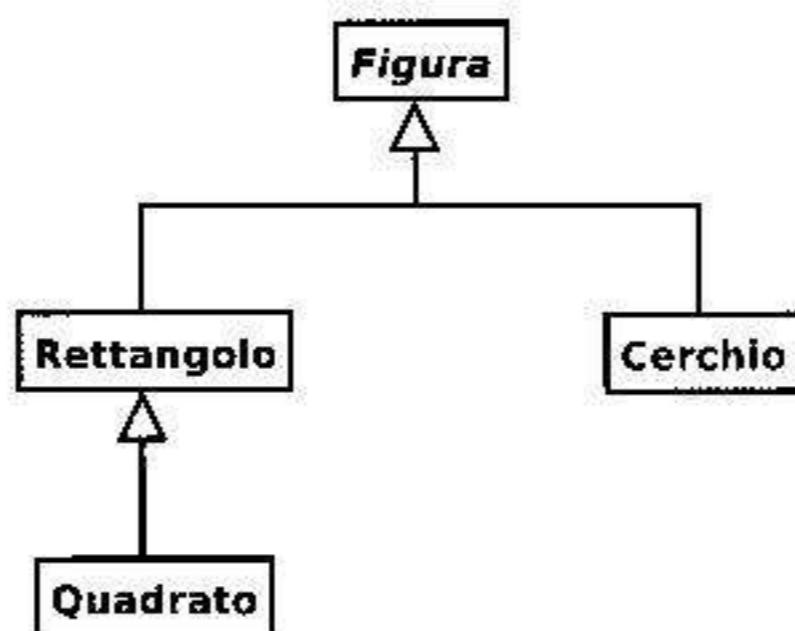


Figura 6.8 La gerarchia della classe Figura.

dichiarate di tipo **Figura**. Per il resto, a parte il menu di scelta, il codice del metodo **main** è molto simile a quello della versione precedente. Si noti anche un piccolo cambiamento nella fase di comunicazione del risultato, con l'uso dell'operatore **instanceof** per determinare il tipo della figura ottenuta.

```

import prog.io.*;
import prog.utili.Figura;
import prog.utili.Rettangolo;
import prog.utili.Quadrato;
import prog.utili.Cerchio;

class FiguraAreaMax {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        Figura r = null, rAreaMax = null;
        boolean continuare = true;
        double x, y;
        char scelta;

        do {
            //scelta della figura
            out.println("      Scelte disponibili: ");
            out.println();
            out.println(" r  Inserimento di un rettangolo");
            out.println(" q  Inserimento di un quadrato");
  
```

```
out.println(" c Inserimento di un cerchio");
out.println(" f Fine inserimento");
out.println();
scelta = in.readChar("    Scelta? ");
out.println("");

//legge i dati di una figura del tipo selezionato
boolean inseritaFigura = false;
switch (scelta) {
case 'r':
    out.println("Inserimento di un rettangolo:");
    x = in.readDouble("    base? ");
    y = in.readDouble("    altezza? ");
    if (x < 0 || y < 0)
        out.println("I dati inseriti non rappresentano un " +
                    "rettangolo");
    else {
        r = x == y ? new Quadrato(x) : new Rettangolo(x, y);
        inseritaFigura = true;
    }
    break;
case 'q':
    out.println("Inserimento di un quadrato:");
    x = in.readDouble("    lato? ");
    if (x < 0)
        out.println("Il dato inserito non può essere il lato " +
                    "di un quadrato");
    else {
        r = new Quadrato(x);
        inseritaFigura = true;
    }
    break;
case 'c':
    out.println("Inserimento di un cerchio");
    x = in.readDouble("    raggio? ");
    if (x < 0)
        out.println("Il dato inserito non può essere il raggio " +
                    "di un cerchio");
    else {
        r = new Cerchio(x);
        inseritaFigura = true;
    }
}
```

```

        break;
    case 'f':
        continuare = false;
        break;
    default:
        out.println(" Scelta non valida\n\n");
        break;
    }
    if (inseritaFigura) {
        out.println(" " + r.toString());
        out.println(" area = " + r.getArea() +
                   ", perimetro = " + r.getPerimetro());

        //confronta la figura con quella di area maggiore
        if (rAreaMax == null || r.haAreaMaggiorer(rAreaMax))
            rAreaMax = r;
        }
        out.println();
    } while (continuare);

    //comunica le caratteristiche della figura di area maggiore
    if (rAreaMax == null)
        out.println("Non è stata inserita alcuna figura");
    else {
        out.print("La figura di area maggiore è un ");
        if (rAreaMax instanceof Cerchio)
            out.print("cerchio: ");
        else if (rAreaMax instanceof Quadrato)
            out.print("quadrato: ");
        else
            out.print("rettangolo: ");
        out.println(" " + rAreaMax.toString());
        out.println(" area = " + rAreaMax.getArea() +
                   ", perimetro = " + rAreaMax.getPerimetro());
    }
}
}

```

Della classe **Figura** abbiamo detto poco: sappiamo che le sue istanze rappresentano figure geometriche sul piano e conosciamo alcune sue sottoclassi. Tuttavia non abbiamo accennato nulla sui costruttori e sui metodi che offre. In realtà, scrivendo il codice precedente abbiamo sottinteso alcuni aspetti inerenti ai metodi offerti dalla classe **Figura**, che ora è bene esplicitare.

Osserviamo che tramite i riferimenti **r** e **rAreaMax** di tipo **Figura** sono stati chiamati i me-

todi `toString`, `getArea`, `getPerimetro` e `haAreaMaggiore`. Affinché la compilazione della classe `FiguraAreaMax` avvenga correttamente, è *necessario* che la classe `Figura` contenga tali metodi. Tuttavia la classe `Figura` non è in grado di implementarli. Ad esempio, sebbene l'area e il perimetro siano due grandezze associabili a tutte le figure sul piano (cioè ogni figura ha un'area e un perimetro), il procedimento per calcolarle dipende dal tipo di figura considerata. Dunque chi progetta la classe `Figura` sarà costretto a indicare l'esistenza di alcuni metodi, come `getArea` e `getPerimetro` (in quanto fanno parte del *comportamento* di ogni figura), rimandandone però l'implementazione alle sottoclassi. Questi metodi prendono il nome di metodi *astratti*. Una classe contenente metodi *astratti* è detta *classe astratta*. Una classe astratta *non può essere istanziata*, ma può essere estesa.¹ Le sottoclassi di una classe astratta dovranno contenere l'implementazione di tutti i metodi astratti, salvo che siano anch'esse astratte. Le classi che non sono astratte, cioè quelle che abbiamo incontrato finora, sono anche chiamate *classi concrete*.

Possono essere implementati direttamente nella classe `Figura` altri metodi, ed è opportuno che lo siano. Ad esempio il metodo `haAreaMaggiore` dovrà essere in grado di confrontare due figure qualsiasi. Pertanto andrà collocato nella classe `Figura` con l'intestazione

```
public boolean haAreaMaggiore(Figura altra)
```

In questo modo, poiché `Rettangolo`, `Quadrato` e `Cerchio` sono sottotipi di `Figura`, è possibile passare come argomento al metodo un riferimento a un oggetto di una di queste classi. Osserviamo che il metodo può essere implementato direttamente nella classe `Figura` in quanto il procedimento da utilizzare non dipende dal tipo di figura considerata. Infatti esso consiste nel calcolo e nel confronto delle aree. Il modo di calcolare l'area cambia in base al tipo di figura, ma ciò può essere delegato al metodo `getArea`. Tali aspetti saranno ulteriormente analizzati studiando l'implementazione delle classi. Una volta calcolate le aree delle due figure, per fornire il risultato il metodo deve semplicemente fare un confronto tra queste due quantità.

In UML le classi astratte sono rappresentate come le classi concrete, con la sola differenza che il loro nome viene scritto in carattere italico. La stessa notazione è adottata per i metodi astratti. Ad esempio la classe astratta `Figura`, con i metodi astratti `getPerimetro` e `getArea` e con i metodi concreti `haAreaMaggiore`, `haPerimetroMaggiore` e `equals`, viene presentata come nella Figura 6.9.

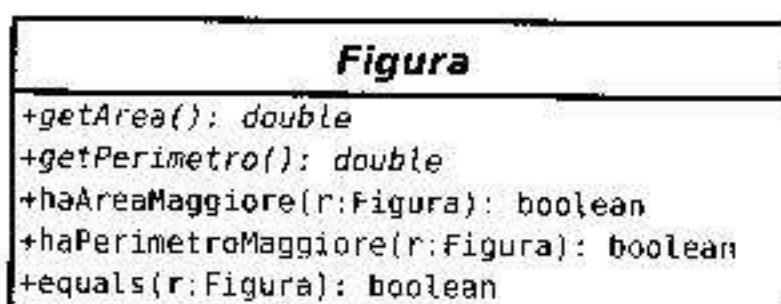


Figura 6.9 La classe astratta `Figura`.

¹ Anche se non può essere istanziata, una classe astratta possiede costruttori che non possono tuttavia essere invocati con espressioni `new`. L'utilità di questi costruttori risulterà chiara quando studieremo l'implementazione delle classi, in particolare quando vedremo come vengono costruiti gli oggetti. Inoltre osserviamo, che allo scopo di impedire che vengano istanziate, è possibile definire classi astratte anche se esse non possiedono metodi astratti.

Come evidenziato nell'esempio precedente, le classi astratte, oltre a fornire un meccanismo per raggruppare in una struttura gerarchica classi logicamente correlate, risultano utili perché forniscono un supertipo comune a tutte le loro sottoclassi: sebbene la classe astratta non possa essere istanziata, il suo tipo può essere usato come riferimento per gli oggetti delle sottoclassi. Nel caso particolare preso in esame, il tipo Figura è indispensabile per definire variabili (`r` o `rAreaMax`) in grado di riferire oggetti di sottoclassi differenti.

Quadr.
mento
Figura
giore.

6.6

Tutte le
quale v
indirett

In p
classe c
estenda

Ad
Quadra
de (ind
Figura
e di Fr
Figura

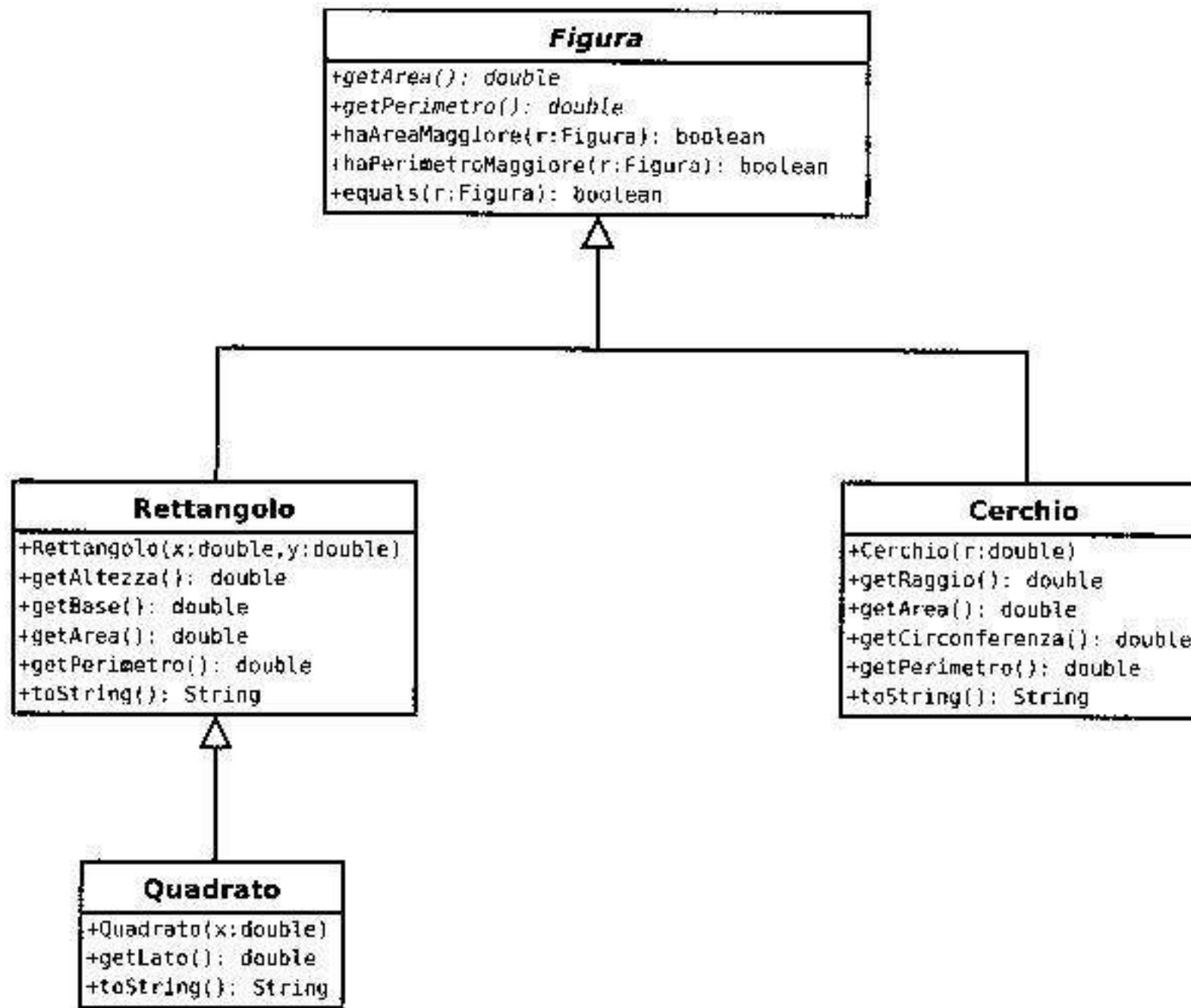


Figura 6.10 La classe astratta Figura e alcune sottoclassi.

Concludiamo questo paragrafo indicando nella Figura 6.10 la collocazione di alcuni metodi che abbiamo incontrato nelle classi utilizzate in questi esempi. In particolare sono elencati accanto al nome di ogni classe i metodi in essa definiti, con la specificazione del prototipo.

Quando in una sottoclasse c'è un metodo con la stessa segnatura di uno della superclasse, il metodo viene ridefinito (ad esempio il metodo `toString` della classe **Rettangolo** è ridefinito nella classe **Quadrato**). Questo significa che il metodo, cioè la sequenza di operazioni con cui un oggetto della sottoclasse risponde al messaggio, è diverso da quello della superclasse. Se invece un metodo della superclasse non è indicato nella sottoclasse, allora la sottoclasse eredita il metodo dalla superclasse (ad esempio **Quadrato** eredita il metodo `getArea` da **Rettangolo** e il metodo `haAreaMaggiori` da **Figura**).

Precisiamo che esiste un unico metodo `haAreaMaggiori`. Esso è definito nella classe **Figura** e riceve un parametro di tipo **Figura**. Poiché **Figura** è un supertipo di **Rettangolo**,

Nel ling
classe p
molte c
diretta.
struttura

Quadrato e Cerchio, il metodo è in grado di ricevere anche parametri di questi tipi. Al momento della chiamata, il riferimento del sottotipo sarà automaticamente promosso al supertipo Figura. Pertanto non c'è bisogno di definire i metodi haAreaMaggiore e haPerimetroMaggiore, che avevamo indicato in precedenza, nelle classi Rettangolo, Quadrato e Cerchio.

6.6 La gerarchia delle classi

Tutte le classi definibili nel linguaggio Java si trovano all'interno di una gerarchia, in testa alla quale vi è una classe predefinita denominata `Object`. Ogni classe Java estende, direttamente o indirettamente, la classe `Object`.

In particolare, quando nell'intestazione di una classe non ci sono indicazioni esplicite sulla classe che essa estende (cioè non viene utilizzata la parola chiave `extends`), è sottinteso che estenda direttamente `Object`.

Ad esempio le classi `Figura` e `Frazione` estendono (direttamente) `Object`. La classe `Quadrato` estende `Rettangolo` che, a sua volta, estende `Figura`. Quindi `Quadrato` estende (indirettamente) `Figura` e `Object`. Quindi, `Quadrato` è una sottoclasse di `Object`, di `Figura` e di `Rettangolo`. `Object` è una superclasse di `Quadrato`, di `Rettangolo`, di `Figura` e di `Frazione`. La parte di gerarchia delle classi che stiamo considerando è illustrata nella Figura 6.11.

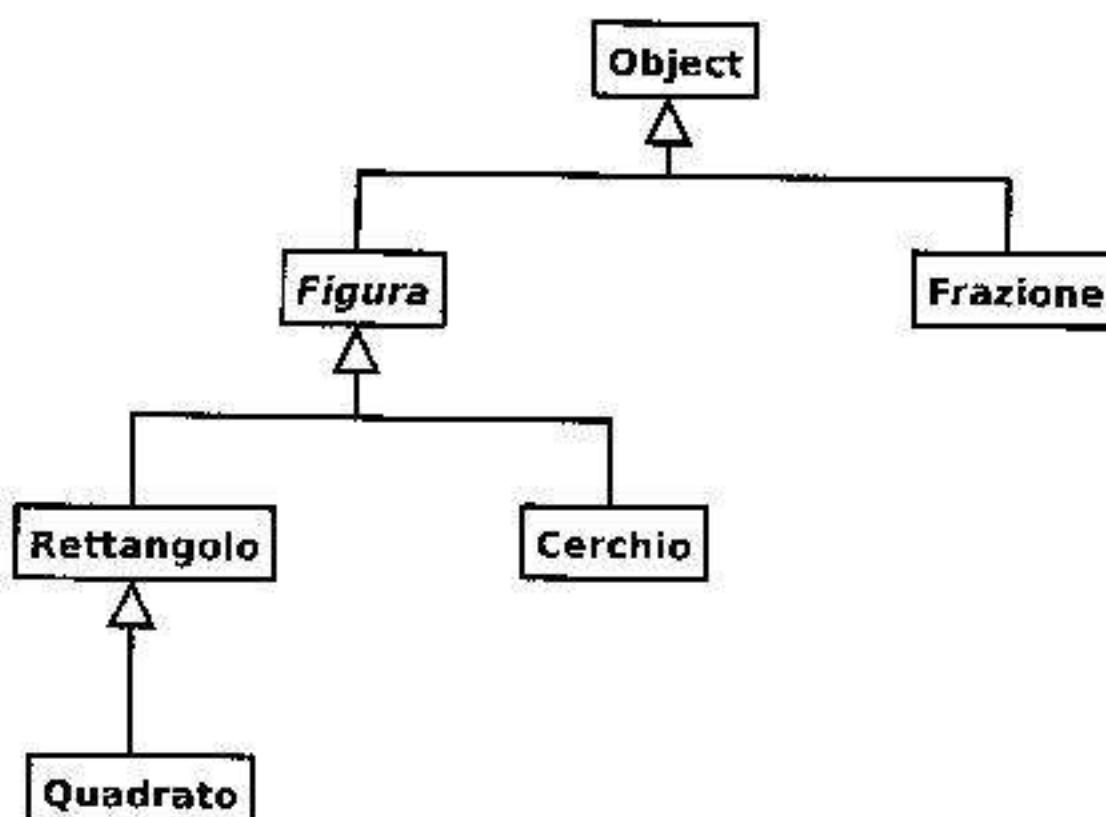


Figura 6.11 La gerarchia delle classi.

Nel linguaggio Java una classe può ereditare al massimo da una classe. Questo significa che ogni classe può avere tutt'alpiù una superclasse diretta. Al contrario, una classe può essere estesa da molte classi; possono dunque esserci molte classi che hanno in comune la medesima superclasse diretta. Pertanto la gerarchia delle classi di Java (Figura 6.11) può essere rappresentata con una struttura ad albero, alla cui radice si trova la classe `Object`.

La classe `Object` contiene alcuni metodi (che quindi vengono ereditati da tutte le altre classi). Tra questi segnaliamo:

- `equals`, che riceve un parametro `Object` e restituisce un risultato `boolean`. Restituisce `true` se e solo se l'oggetto che esegue il metodo è *lo stesso oggetto* fornito tramite il parametro. Si noti che questo metodo `equals` è del tutto equivalente all'operatore di confronto `==` applicato a riferimenti. Al fine di controllare l'uguaglianza tra ciò che due oggetti rappresentano è sempre opportuno che nelle sottoclassi di `Object` tale metodo sia ridefinito.²
- `toString` che, privo di parametri, restituisce il riferimento a un oggetto di tipo `String`, il quale rappresenta alcune informazioni poco comprensibili relative all'oggetto che lo esegue. È sempre opportuno che questo metodo sia ridefinito nelle sottoclassi.

La classe `Object` ha inoltre un costruttore, privo di argomenti, che si occupa solo di creare l'istanza dell'oggetto.

6.7 Gerarchia e uso dei riferimenti

Come abbiamo già osservato, è possibile assegnare a una variabile, il cui tipo sia una classe, un riferimento a un oggetto di una sottoclasse. Ad esempio possiamo assegnare a una variabile di tipo `Rettangolo` un riferimento a un'istanza della classe `Quadrato`. Poiché `Object` è una superclasse di tutte le classi, possiamo assegnare alle variabili di tipo `Object` riferimenti a oggetti di qualunque classe. Viceversa, non è possibile assegnare direttamente a una variabile che corrisponde a una sottoclasse un riferimento a un oggetto di una superclasse, ma è necessario utilizzare un cast.

Ad esempio, avendo dichiarato

```
Rettangolo r;
Quadrato q;
```

l'assegnamento

```
r = q;
```

è consentito, mentre non lo è l'assegnamento

```
q = r;
```

Tuttavia in questo caso è possibile utilizzare un cast esplicito alla sottoclasse

```
q = (Quadrato) r;
```

² La classe `Object` possiede anche un metodo `hashCode` che è opportuno ridefinire nelle sottoclassi, affinché queste possano essere correttamente utilizzate all'interno di collezioni indirizzate. L'argomento richiede conoscenze ben più approfondate di quelle fornite sin qui ed esula dagli scopi di questo testo, pertanto si rimanda il lettore al manuale ufficiale del linguaggio.

La variabile
un'istanza
getto ri-
precisa
da un co-

if (C
q
else

Dato ch
riferime
riferime
riferime
no: il co
quanto c
non avre

6.8

Abbiam
Il metod
riscritte
di overrid
esecuzio
del rifer
di almen

Con

Obje
Rett
Quad

Poiché
utilizzat

o =

oppure:

o =

per la cu

La variabile `r` è un riferimento a un'istanza della classe `Rettangolo`. Se tale oggetto è anche un'istanza della classe `Quadrato`, allora il cast è eseguito senza problemi. Viceversa, se l'oggetto riferito da `r` non è un'istanza di `Quadrato`, il cast provoca un errore in esecuzione (o, più precisamente, un'eccezione). È pertanto opportuno che un cast come il precedente sia preceduto da un controllo, mediante l'uso dell'operatore `instanceof`, del tipo:

```
if (r instanceof Quadrato)
    q = (Quadrato) r;
else ...
```

Dato che `Object` è una superclasse di tutte le classi, e dunque `Object` è un supertipo di tutti i tipi riferimento, potremmo essere tentati di utilizzare il tipo `Object` tutte le volte che serve un tipo riferimento. Nell'ultima versione di `FiguraAreaMax`, avremmo ad esempio potuto dichiarare i riferimenti `r` e `rAreaMax` di tipo `Object` evitando di utilizzare il tipo `Figura`? La risposta è no: il compilatore non avrebbe accettato le invocazioni dei metodi `area` e `haAreaMaggior`, in quanto essi non sono disponibili per `Object`. L'invocazione del metodo `toString`, al contrario, non avrebbe dato problemi, poiché esso è disponibile anche in questa classe.

6.8 Scelta del metodo da eseguire

Abbiamo visto come sia possibile riscrivere metodi delle superclassi all'interno delle sottoclassi. Il metodo della sottoclasse ha la medesima segnatura di quello della superclasse che è stato riscritto. La riscrittura dei metodi è chiamata *overriding* (letteralmente "sovrascrittura"). In caso di overriding, il metodo da eseguire viene determinato dalla Java Virtual Machine in fase di esecuzione *sulla base del tipo effettivo dell'oggetto* che esegue il metodo (e non in base al tipo del riferimento). Il tipo del riferimento è invece utilizzato dal compilatore per stabilire l'esistenza di almeno un metodo associabile alla chiamata.

Consideriamo, ad esempio, le seguenti dichiarazioni:

```
Object o;
Rettangolo r;
Quadrato q;
```

Poiché `Object` è una superclasse di `Rettangolo` e di `Quadrato`, il riferimento `o` può essere utilizzato anche per istanze di queste due classi. Possiamo cioè scrivere assegnamenti come:

```
o = r;
```

oppure:

```
o = new Quadrato(10);
```

per la cui esecuzione saranno effettuate promozioni implicite.

Tramite il riferimento o possiamo inoltre invocare metodi associati agli oggetti. Ad esempio possiamo scrivere:

`o.toString()`

per invocare il metodo `toString` dell'oggetto riferito da `o`. In fase d'esecuzione, se l'oggetto è un'istanza di `Quadrato`, sarà eseguito il metodo della classe `Quadrato`; se l'oggetto è un'istanza di `Rettangolo`, ma non di `Quadrato`, sarà eseguito quello di `Rettangolo`; se l'oggetto è un'istanza solo di `Object`, sarà eseguito il metodo di `Object`. Ciò che accade nei tre casi è riassunto nella Figura 6.12: in fase di esecuzione la Java Virtual Machine cerca il codice da eseguire nella classe di cui è istanza l'oggetto. Ad esempio, nel caso in cui `o` faccia riferimento a un'istanza della classe `Quadrato` (primo caso nella Figura 6.12), la Java Virtual Machine inizia a cercare il metodo nel codice della classe `Quadrato` (più precisamente nel file `Quadrato.class` prodotto dal compilatore). Trovato il metodo, lo manda in esecuzione.

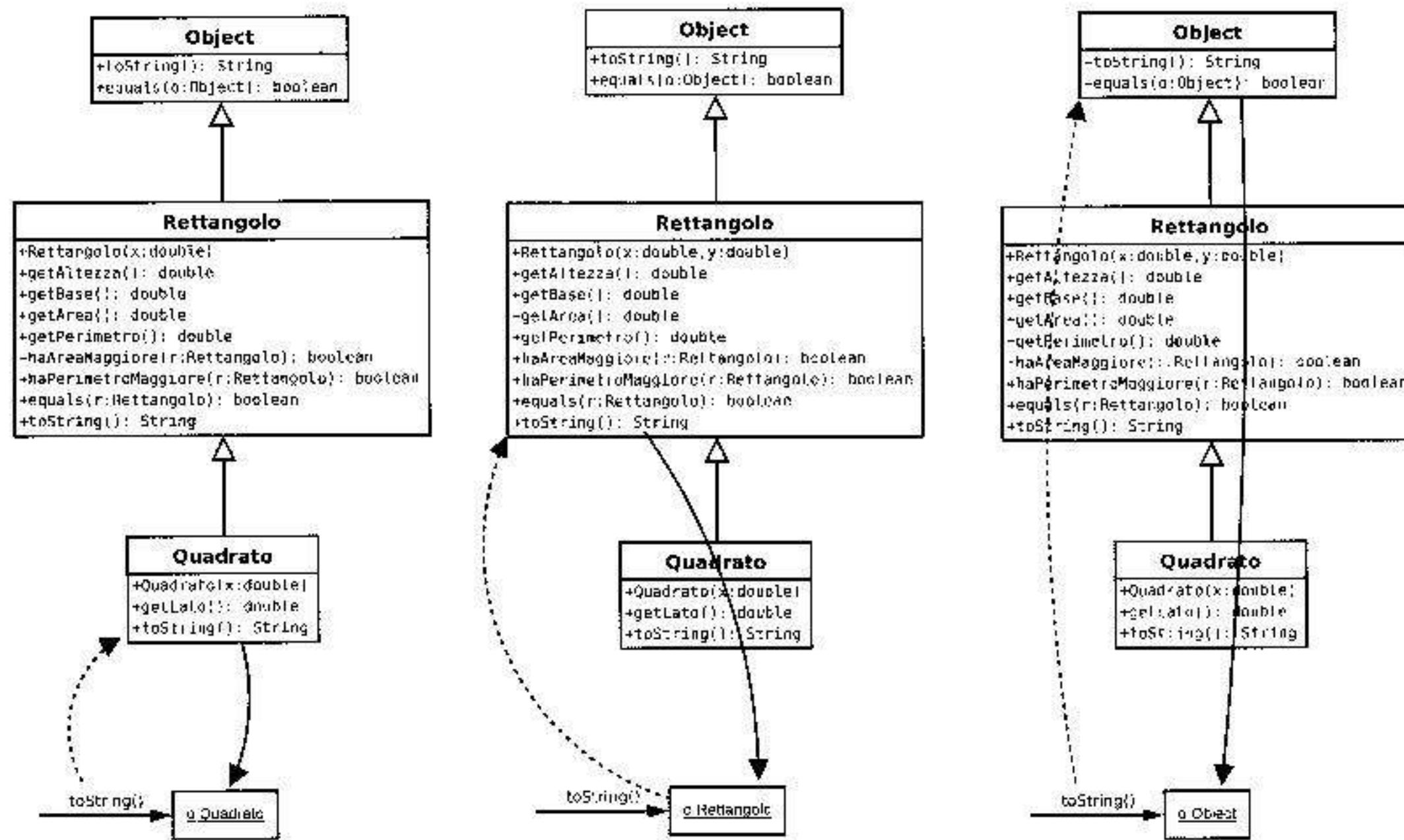


Figura 6.12 Ricerca del metodo in fase di esecuzione.

Il secondo e il terzo caso nella Figura 6.12 sono analoghi al primo: la Java Virtual Machine trova sempre il metodo `toString` da eseguire nella classe da cui inizia la ricerca.

Supponiamo ora di scrivere la chiamata

`o.getBase()`

Questa potrebbe essere corretta se l'oggetto riferito da `r` fosse un'istanza di `Rettangolo`. Tuttavia, se l'oggetto fosse un'istanza solo di `Object`, l'invocazione sarebbe scorretta perché in fase di esecuzione la Java Virtual Machine non sarebbe in grado di trovare un metodo con cui rispondere al messaggio. Il compilatore, non conoscendo l'oggetto cui farà effettivamente riferimento `r` durante l'esecuzione, segnalerà pertanto un errore in modo da evitare possibili anomalie in fase di esecuzione.

Analizzando un'invocazione di metodo, il compilatore controlla che, per il tipo utilizzato come riferimento, *esista almeno un metodo che possa essere invocato*.

La chiamata

```
r.getBase()
```

è compilata correttamente, perché per la classe `Rettangolo` esiste un metodo `getBase` privo di argomenti. Invece la chiamata

```
r.getLato()
```

non viene compilata, perché la classe `Rettangolo` non dispone di un metodo `getLato`.

Si decide quale metodo eseguire solo in fase di esecuzione, osservando il tipo dell'oggetto. In particolare, il metodo da eseguire viene determinato risalendo nella gerarchia delle classi a partire dalla classe effettiva dell'oggetto.

Consideriamo le seguenti linee di codice:

```
Rettangolo r;
int x, y;
...
if (x == y)
    r = new Quadrato(x);
else
    r = new Rettangolo(x, y);
System.out.println(r.toString());
System.out.println(r.getBase());
```

Le righe sono compilate correttamente, perché per il tipo di `r` esistono ambedue i metodi `toString` e `getBase`. Analizziamo ora due esecuzioni differenti.

- Se `x` e `y` contengono entrambe inizialmente 3, viene creata un'istanza della classe `Quadrato` che rappresenta un quadrato di lato 3. È poi invocato il metodo `toString`. Come spiegato sopra, tale metodo è ricercato a partire dalla classe effettiva dell'oggetto, cioè da `Quadrato`. Poiché tale classe possiede un proprio metodo `toString`, esso viene eseguito, restituendo così la stringa "lato = 3" che sarà poi visualizzata. Nella riga successiva, è visualizzato il risultato del metodo `getBase`. Anche in questo caso il metodo viene individuato a partire dalla classe dell'oggetto effettivamente riferito da `r`, cioè da `Quadrato`. Chi ha implementato la classe `Quadrato` non ha in realtà definito il metodo `getBase` all'interno di questa, lasciando che esso venisse ereditato dalla superclasse. Pertanto, non essendovi il metodo in `Quadrato`, la ricerca prosegue nella superclasse `Rettangolo`, dove il metodo è individuato ed eseguito. Questo caso è descritto nella Figura 6.13.

- Se `x` contiene inizialmente 10 e `y` contiene inizialmente 8, viene costruita un’istanza di `Rettangolo`. In questo caso la ricerca dei metodi `toString` e `getBase` da eseguire avverrà durante l’esecuzione a partire dalla classe `Rettangolo`, in cui saranno individuati ambedue i metodi.

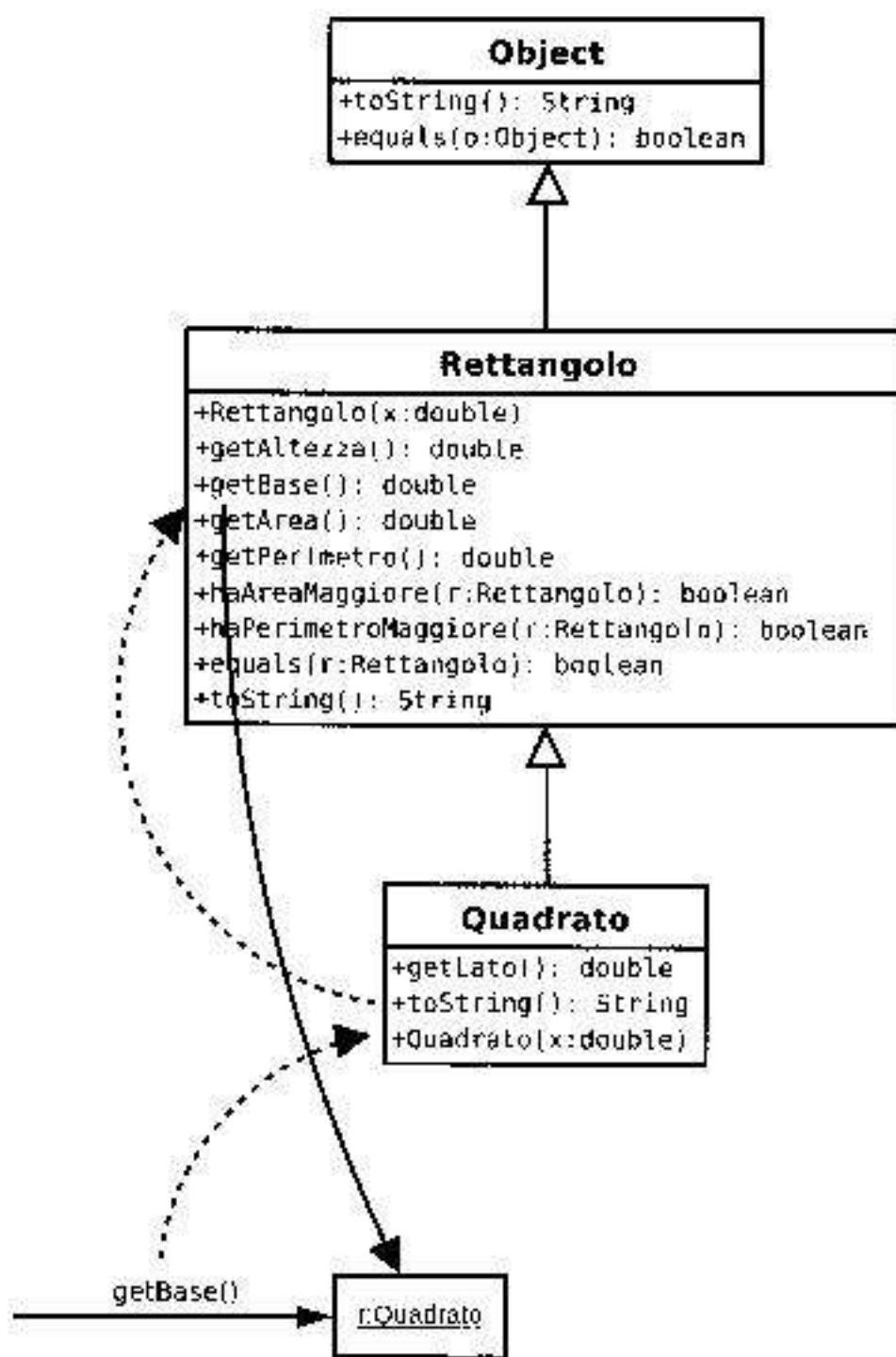


Figura 6.13 Ricerca del metodo in fase di esecuzione.

Riepilogando, possiamo affermare che la scelta del metodo da eseguire avviene in due passi fondamentali.

- (1) In *fase di compilazione* si verifica l’esistenza, per il tipo del riferimento utilizzato, di un metodo che soddisfi la chiamata; in questa fase è risolto anche l’eventuale overloading, scegliendo la segnatura del metodo che dovrà essere eseguito.
- (2) In *fase di esecuzione* viene selezionato il metodo da eseguire, sulla base del tipo effettivo dell’oggetto e non del tipo del riferimento; in particolare viene ricercato un metodo che abbia la segnatura selezionata durante la fase di compilazione. La ricerca avviene a partire dalla classe dell’oggetto, risalendo progressivamente nella gerarchia fino a trovare il metodo desiderato. Poiché il compilatore ha controllato l’esistenza di un tale metodo per il tipo

del riferimento, prima o poi il metodo selezionato sarà trovato (al massimo risalendo nella gerarchia fino al tipo del riferimento).

Nel Capitolo 8, studiando superclassi e sottoclassi nell'ottica dell'implementazione, approfondiremo questi aspetti e analizzeremo ciò che accade quando si verificano contemporaneamente overloading e overriding.

Esercizi

6.1 Considerate le seguenti dichiarazioni:

```
Object o;  
Rettangolo r;  
Quadrato q;  
Frazione f;
```

Per ognuna delle seguenti righe di codice individuate eventuali errori in compilazione. In caso negativo, nell'ipotesi che le quattro variabili non contengano null, individuate eventuali errori in esecuzione e indicate una situazione in cui ciò avviene spiegandone il motivo. Indicate inoltre quali cast sono inutili.

- (1) `o.toString();`
- (2) `r.toString();`
- (3) `q.toString();`
- (4) `f.toString();`
- (5) `o = o;`
- (6) `o = r;`
- (7) `o = q;`
- (8) `o = f;`
- (9) `r = o;`
- (10) `r = q;`
- (11) `r = f;`
- (12) `o = (Rettangolo) r;`
- (13) `r = (Rettangolo) r;`
- (14) `q = (Rettangolo) r;`
- (15) `f = (Rettangolo) r;`
- (16) `o = (Quadrato) r;`
- (17) `r = (Quadrato) r;`

- (18) q = (Quadrato) r;
- (19) f = (Quadrato) r;
- (20) r = (Rettangolo) o;
- (21) r = (Rettangolo) q;
- (22) r = (Rettangolo) f;
- (23) f = (Frazione) o;
- (24) o = (Frazione) f;

6.2 Per ognuna delle seguenti porzioni di codice indicate quale metodo `toString` sarà eseguito o se c'è un errore (specificando, in questo caso, se in compilazione o in esecuzione). Verificate poi le risposte inserendo le linee di codice in brevi programmi di prova.

- (1) Object o = new Quadrato(1);
String s = o.toString();
- (2) Rettangolo o = new Quadrato(1);
String s = o.toString();
- (3) Rettangolo o = new Rettangolo(2, 3);
String s = o.toString();
- (4) Rettangolo o = new Rettangolo(1, 1);
String s = o.toString();
- (5) Rettangolo o = new Quadrato(1);
Quadrato q = (Quadrato) o;
String s = q.toString();
- (6) Rettangolo o = new Rettangolo(2, 3);
Quadrato q = (Quadrato) o;
String s = q.toString();
- (7) Rettangolo o = new Rettangolo(1, 1);
Quadrato q = (Quadrato) o;
String s = q.toString();
- (8) Object o = new Frazione(2, 3);
String s = o.toString();
- (9) Object o = new Frazione(2, 3);
Quadrato q = (Quadrato) o;
String s = q.toString();
- (10) Object o = new Object();
String s = o.toString();

6.3 Indicate l'output ottenuto eseguendo la seguente classe:

6.9

Negli usare paragi Sv legger elenco ne Fi selezi quella costru trario, memo produz zerem perne to dell classi oggett e lo in Seq La fas za rile

```

import prog.utili.Rettangolo;
import prog.utili.Quadrato;

class Prova {
    public static void main(String[] args) {
        Rettangolo r = null;
        System.out.println(r instanceof Rettangolo);
        System.out.println(r instanceof Quadrato);
        r = new Quadrato(1);
        System.out.println(r instanceof Rettangolo);
        System.out.println(r instanceof Quadrato);
        r = new Rettangolo(1, 1);
        System.out.println(r instanceof Rettangolo);
        System.out.println(r instanceof Quadrato);
    }
}

```

6.9 Esempio: gestione di un elenco di figure

Negli esempi dei paragrafi precedenti abbiamo evidenziato come sia possibile (e spesso utile) usare tipi riferimento corrispondenti alle superclassi per riferirsi a oggetti di sottoclassi. In questo paragrafo illustriamo un ulteriore esempio di quest'uso considerando collezioni di oggetti.

Svilupperemo un'applicazione, che chiameremo **ElencaFigure**, il cui compito consiste nel leggere i dati relativi a una sequenza di figure (rettangoli, quadrati e cerchi) e nel produrre un elenco di tutte le figure inserite, suddiviso per tipi. L'applicazione, analogamente all'applicazione **FiguraAreaMax** sviluppata nei paragrafi precedenti, deve fornire all'utente la possibilità di selezionare la figura da inserire mediante un menu. La parte di acquisizione dei dati è simile a quella dell'applicazione **FiguraAreaMax**, in cui, acquisendo i dati relativi a una figura, viene costruita un'istanza della classe corrispondente. La modalità di elaborazione dei dati è, al contrario, molto diversa: abbiamo visto che per determinare la figura con area maggiore non occorre memorizzare l'elenco completo delle figure inserite; in questo caso, invece, è evidente che per produrre il risultato richiesto la sua memorizzazione è indispensabile. A questo scopo utilizzeremo un'istanza della classe generica **Sequenza**. Ricordiamo che le istanze di questa classe permettono di memorizzare sequenze di oggetti di uno *stesso tipo*, specificato come argomento della classe stessa. Gli oggetti che dobbiamo memorizzare nella sequenza sono istanze delle classi **Rettangolo**, **Quadrato** e **Cerchio**. Di conseguenza, possiamo utilizzare una sequenza di oggetti del supertipo comune, cioè **Figura**. Chiamiamo **figure** il riferimento a tale sequenza, e lo inizializziamo associandogli la sequenza vuota:

```
Sequenza<Figura> figure = new Sequenza<Figura>();
```

La fase di lettura dei dati ricalca quella dell'applicazione **FiguraAreaMax**. L'unica differenza rilevante è che, dopo avere letto una figura, essa dovrà essere memorizzata nella sequenza.

In particolare, detta `r` la variabile di tipo `Figura` che contiene il riferimento alla figura letta, l'istruzione

```
figure.add(r);
```

permette di aggiungere la figura alla sequenza.

Terminata la fase di lettura, l'applicazione deve produrre l'elenco richiesto. A tale scopo, per individuare le informazioni da visualizzare, è sufficiente scandire la sequenza, una volta per ciascun tipo di figura. Per la scansione ricorriamo a un ciclo `for-each` che esamina via via ciascun elemento. Ad esempio, per produrre l'elenco dei cerchi utilizzeremo questo schema:

```
for (Figura f : figure)
    if (l'oggetto riferito da f rappresenta un cerchio)
        visualizza le informazioni relative all'oggetto riferito da f
```

La condizione della selezione può essere scritta utilizzando l'operatore `instanceof`. Inoltre, per ottenere la stringa contenente le informazioni da visualizzare, possiamo richiedere all'oggetto di eseguire il proprio metodo `toString` scrivendo `f.toString()`. In questo caso, grazie al polimorfismo, il metodo selezionato sarà quello della classe `Cerchio`. Ecco la codifica basata sullo schema precedente:

```
for (Figura f : figure)
    if (f instanceof Cerchio)
        out.println(f.toString());
```

Per visualizzare i quadrati si procede in maniera simile. Nel caso dei rettangoli è necessario introdurre un ulteriore controllo, per evitare di visualizzare insieme a questi anche i quadrati:

```
for (Figura f : figure)
    if (f instanceof Rettangolo &&
        !(f instanceof Quadrato))
        out.println(f.toString());
```

Segue il codice completo dell'applicazione così sviluppata:

```
import prog.io.*;
import prog.utili.Figura;
import prog.utili.Rettangolo;
import prog.utili.Quadrato;
import prog.utili.Cerchio;
import prog.utili.Sequenza;

class ElencaFigure {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
```

```
ConsoleInputManager in = new ConsoleInputManager();
ConsoleOutputManager out = new ConsoleOutputManager();

Sequenza<Figura> figure = new Sequenza<Figura>();

Figura r = null;
boolean continuare = true;
double x, y;
char scelta;

do {
    //scelta della figura
    out.println(" Scelte disponibili: ");
    out.println();
    out.println(" r Inserimento di un rettangolo");
    out.println(" q Inserimento di un quadrato");
    out.println(" c Inserimento di un cerchio");
    out.println(" f Fine inserimento");
    out.println();
    scelta = in.readChar(" Scelta? ");
    out.println("");

    //legge i dati di una figura del tipo selezionato
    boolean inseritaFigura = false;
    switch (scelta) {
        case 'r':
            out.println("Inserimento di un rettangolo:");
            x = in.readDouble(" base? ");
            y = in.readDouble(" altezza? ");
            if (x < 0 || y < 0)
                out.println("I dati inseriti non rappresentano un " +
                           "rettangolo");
            else {
                r = x == y ? new Quadrato(x) : new Rettangolo(x, y);
                inseritaFigura = true;
            }
            break;
        case 'q':
            out.println("Inserimento di un quadrato:");
            x = in.readDouble(" lato? ");
            if (x < 0)
                out.println("Il dato inserito non può essere il lato " +
```

```

        "di un quadrato");
    else {
        r = new Quadrato(x);
        inseritaFigura = true;
    }
    break;
case 'c':
    out.println("Inserimento di un cerchio");
    x = in.readDouble("    raggio? ");
    if (x < 0)
        out.println("Il dato inserito non può essere il raggio " +
                    "di un cerchio");
    else {
        r = new Cerchio(x);
        inseritaFigura = true;
    }
    break;
case 'f':
    continuare = false;
    break;
default:
    out.println("    Scelta non valida\n\n");
    break;
}
if (inseritaFigura) {
    out.println("    " + r.toString());
    out.println("    area = " + r.getArea() +
               ", perimetro = " + r.getPerimetro());

    //inserisce la figura nella sequenza
    figure.add(r);
}
out.println();
} while (continuare);

//elenco dei cerchi inseriti
out.println("*** ELENCO DEI CERCHI ***");
for (Figura f : figure)
    if (f instanceof Cerchio)
        out.println(f.toString());
out.println();

```

Eserc

6.4 Mo

6.5 Mo

Per
rife
di t
form
tipi
gic
dur

```

//elenco dei quadrati inseriti
out.println("*** ELENCO DEI QUADRATI ***");
for (Figura f : figure)
    if (f instanceof Quadrato)
        out.println(f.toString());
out.println();

//elenco dei rettangoli inseriti
out.println("*** ELENCO DEI RETTANGOLI ***");
for (Figura f : figure)
    if (f instanceof Rettangolo &&
        !(f instanceof Quadrato))
        out.println(f.toString());
out.println();
}
}

```

Esercizi

6.4 Modificate l'applicazione `ElencaFigure` in modo che comunichi:

- il perimetro medio delle figure inserite e le caratteristiche della figura il cui perimetro si avvicina maggiormente a esso, specificando di che tipo di figura si tratta;
- l'area media delle figure inserite e le caratteristiche della figura la cui area si avvicina maggiormente a essa, specificando di che tipo di figura si tratta;
- il numero di quadrati, di rettangoli e di cerchi la cui area è inferiore all'area media;
- le somme delle aree e dei perimetri di tutti i cerchi, di tutti i quadrati e di tutti i rettangoli inseriti.

6.5 Modificate l'applicazione `ElencaFigure` in modo che comunichi:

- la media delle basi e la media delle altezze di tutti i rettangoli inseriti;
- la media dei lati di tutti i quadrati inseriti;
- la media dei raggi di tutti i cerchi inseriti.

Per svolgere l'esercizio tenete conto della trattazione relativa alla gerarchia e all'uso dei riferimenti. In particolare, osservate che i riferimenti presenti nella sequenza `figure` sono di tipo `Figura`: utilizzandoli direttamente non è pertanto possibile invocare i metodi che forniscono i dati necessari per risolvere il problema. Ad esempio, se `f` è una variabile di tipo `Figura`, dichiarata per la scansione in un ciclo `for-each`, l'invocazione `f.getRaggio()` non è lecita perché per il tipo `Figura` il metodo `getRaggio` non esiste. Tuttavia, se durante l'esecuzione abbiamo verificato che l'oggetto riferito da `f` è un cerchio, possiamo

forzare il riferimento al tipo `Cerchio` mediante un cast e ottenere un nuovo riferimento allo stesso oggetto, utilizzabile per invocare il metodo:

```
...dopo avere verificato che f è un riferimento  
a un cerchio...  
Cerchio c = (Cerchio)f;  
...c.getRaggio()...
```

- 6.6 Riscrivete l'applicazione `ElencaFigure` utilizzando, per memorizzare le figure lette, un array di oggetti di tipo `Figura`, al posto dell'oggetto di tipo `Sequenza<Figura>`.

6.10 I file di testo

Un *file* è un archivio di dati che si trova nella memoria di massa. Molte applicazioni utilizzate comunemente manipolano masse di dati organizzate in strutture dette *basi di dati*.

Non tratteremo le organizzazioni degli archivi e delle basi di dati, ma ci limiteremo a presentare un particolare tipo di file impiegato per memorizzare testi. Questi file sono denominati *file di testo*. Ad esempio i testi sorgenti dei programmi vengono forniti al compilatore sotto forma di file di testo.

Un file di testo non è altro che un archivio organizzato come una sequenza di righe in cui ciascuna riga è una stringa, cioè una sequenza di caratteri.

Nel package `prog.io` vengono fornite due classi per la manipolazione dei file di testo: la classe `FileInputManager` e la classe `FileOutputManager`.

Per la lettura del file di testo possiamo utilizzare la classe `FileInputManager`. Essa fornisce il seguente costruttore:

- `public FileInputManager(String nomefile)`
Crea un canale di comunicazione per la lettura dal file il cui nome è specificato come argomento. Se il file specificato non esiste, si verifica un errore in fase di esecuzione.

Per evitare che un'applicazione termini prematuramente in seguito al tentativo di costruire un canale di comunicazione con un file che non esiste, è sempre bene verificare preliminarmente l'esistenza del file. A tale scopo la classe `FileInputManager` mette a disposizione il metodo statico:

- `public static boolean exists(String nomefile)`
Restituisce `true` se esiste un file il cui nome è uguale alla stringa specificata come argomento, `false` in caso contrario.

Il principale metodo della classe è:

- `public String readLine()`
Legge una riga di testo dal file cui fa riferimento l'oggetto che esegue il metodo e la restituisce come risultato. Nel caso sia stata raggiunta la fine del file, restituisce `null`.

Un file è un archivio di dati che si trova nella memoria di massa. Molte applicazioni utilizzate comunemente manipolano masse di dati organizzate in strutture dette *basi di dati*. Non tratteremo le organizzazioni degli archivi e delle basi di dati, ma ci limiteremo a presentare un particolare tipo di file impiegato per memorizzare testi. Questi file sono denominati *file di testo*. Ad esempio i testi sorgenti dei programmi vengono forniti al compilatore sotto forma di file di testo.

Un file di testo non è altro che un archivio organizzato come una sequenza di righe in cui ciascuna riga è una stringa, cioè una sequenza di caratteri.

Per evitare che un'applicazione termini prematuramente in seguito al tentativo di costruire un canale di comunicazione con un file che non esiste, è sempre bene verificare preliminarmente l'esistenza del file. A tale scopo la classe `FileInputManager` mette a disposizione il metodo statico:

- `public static boolean exists(String nomefile)`
Restituisce `true` se esiste un file il cui nome è uguale alla stringa specificata come argomento, `false` in caso contrario.

Il principale metodo della classe è:

- `public String readLine()`
Legge una riga di testo dal file cui fa riferimento l'oggetto che esegue il metodo e la restituisce come risultato. Nel caso sia stata raggiunta la fine del file, restituisce `null`.

In maniera automatica.

Un file di testo viene scandito sequenzialmente. Possiamo immaginare che vi sia un “puntatore al file” che indica la prossima riga a partire dalla quale cominciare a leggere. Al momento dell’“apertura” del file, cioè alla chiamata del costruttore, il puntatore è posto sulla prima riga del file. A ogni operazione di lettura, cioè a ogni chiamata di `readLine`, viene restituita la stringa su cui si trova il puntatore, che è spostato automaticamente alla riga successiva. Quando viene raggiunta la fine del file (*end-of-file*), la chiamata di `readLine` restituisce il riferimento `null`. Esistono anche due metodi, `reset` e `close`, privi di argomenti e il cui tipo restituito è `void`: il primo riporta il puntatore all’inizio del file, il secondo chiude il canale di comunicazione.³

Un’applicazione che elabori un file di testo può basarsi sullo schema seguente:

```
crea un canale di comunicazione per la lettura del file
esamina il file
chiudi il file
```

La fase di esame del file si incentrerà su un ciclo in cui si esamina una riga per volta. Il ciclo termina quando si raggiunge la fine del file, cioè quando la chiamata di `readLine` restituisce un riferimento `null` (si osservi che il file potrebbe anche essere vuoto, e dunque il codice interno al ciclo potrebbe essere eseguito zero volte).

Ecco una prima traccia di un metodo `main` che elabora un file di testo:

```
public static void main(String[] args) {
    ...
    //legge il nome del file e crea l'oggetto corrispondente
    String nomeFile = in.readLine("Nome del file da esaminare? " );

    //verifica l'esistenza del file
    if (FileInputManager.exists(nomeFile)) {
        FileInputManager ingresso = new FileInputManager(nomeFile);
        ...
        String riga;
        while ((riga = ingresso.readLine()) != null)
            ...elabora la riga...

        //chiusura del file
        ingresso.close();

        ...eventuali altre operazioni...
    }
    else
        ...comunica che il file non esiste...
}
```

³ In mancanza di una chiamata del metodo `close`, il canale di comunicazione tra un’applicazione e un file viene automaticamente chiuso al termine dell’esecuzione dell’applicazione.

Descriviamo ora brevemente alcuni metodi e costruttori della classe `FileOutputManager`, rimandando alla documentazione per l'elenco completo:

- `public FileOutputManager(String nomefile)`
Crea un canale di comunicazione per la scrittura sul file il cui nome è specificato come argomento. Se il file specificato esiste, il suo contenuto viene cancellato; se non esiste, viene creato.
- `public void println(String s)`
Scrive la stringa fornita come argomento sul file cui fa riferimento l'oggetto che esegue il metodo.⁴
- `public void close()`
Chiude il canale di comunicazione.

Si noti che il costruttore presentato elimina dal file il contenuto precedente.

Utilizzando lo schema di prima, presentiamo ora, senza ulteriori commenti, un'applicazione che copia un file di testo:

```
import prog.io.*;
class CopiaFile {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();

        //legge il nome del file da copiare e crea l'oggetto corrispondente
        String nomeIngresso = in.readLine("Nome del file da copiare? ");
        FileInputStream ingresso = new FileInputStream(nomeIngresso);

        //legge il nome del file su cui copiare e crea l'oggetto
        //corrispondente
        String nomeCopia = in.readLine("Nome della copia? ");
        FileOutputManager uscita = new FileOutputManager(nomeCopia);

        String riga;
        while ((riga = ingresso.readLine()) != null)
            //trascrizione della riga letta
            uscita.println(riga);

        //chiusura dei file
        ingresso.close();
    }
}
```

⁴ Per la scrittura di vari tipi di dati, la classe dispone di metodi denominati `print` e `println`, analoghi a quelli di `ConsoleOutputManager`. I metodi `println`, diversamente dai metodi `print`, completano la riga corrente in modo che la successiva operazione di scrittura inizi alla riga seguente.

```

    uscita.close();
}
}
}
```

Esercizi

- 6.7 Modificate l'applicazione `CopiaFile` in modo che verifichi l'esistenza del file da copiare e segnali, in caso contrario, un errore.
- 6.8 Nel caso esista già un file con lo stesso nome di quello che l'utente indica per la copia, l'applicazione `CopiaFile` cancella il contenuto del file esistente senza fornire alcun messaggio. Modificate l'applicazione in modo che in questo caso chieda conferma del nome inserito all'utente e permetta di indicare un nome diverso.
- 6.9 Modificate l'applicazione `CopiaFile` in modo che riceva il nome del file da copiare e il nome del file in cui effettuare la copia direttamente sulla linea di comando. Se ad esempio i due nomi sono `sorgente` e `destinazione`, la linea di comando sarà:

`java copiaFile sorgente destinazione`

- 6.10 Scrivete una classe che legga un file di testo e lo visualizzi sullo schermo numerando ciascuna riga. Ad esempio, dato il file:

Nel mezzo del cammin di nostra vita
 mi ritrovai per una selva oscura
 che' la diritta via era smarrita.
 Ahi quanto a dir qual era e' cosa dura
 esta selva selvaggia e aspra e forte
 che nel pensier rinnova la paura!
 Tant'e' amara che poco e' piu' morte;
 ma per trattar del ben ch'i' vi trovai,
 diro' de l'altre cose ch'i' v'ho scorte.
 Io non so ben ridir com'i' v'intrai,
 tant'era pien di sonno a quel punto
 che la verace via abbandonai.

si dovrà produrre in output:

- 1 Nel mezzo del cammin di nostra vita
- 2 mi ritrovai per una selva oscura
- 3 che' la diritta via era smarrita.
- 4 Ahi quanto a dir qual era e' cosa dura
- 5 esta selva selvaggia e aspra e forte

6 che nel pensier rinovala paura!
 7 Tant'e' amara che poco e' piu' morte;
 8 ma per trattar del ben ch'i' vi trovai,
 9 diro' de l'altre cose ch'i' v'ho scorte.
 10 Io non so ben ridir com'i' v'intrai,
 11 tant'era pien di sonno a quel punto
 12 che la verace via abbandonai.

- 6.11 Scrivete una classe che legga un file di testo e lo visualizzi sullo schermo, convertendo tutte le lettere minuscole in maiuscole. Ad esempio, l'output prodotto sul file dell'esempio precedente dovrà essere:

NEL MEZZO DEL CAMMIN DI NOSTRA VITA
 MI RITROVAI PER UNA SELVA OSCURA
 CHE' LA DIRITTA VIA ERA SMARRITA.
 AHI QUANTO A DIR QUAL ERA E' COSA DURA
 ESTA SELVA SELVAGGIA E ASPRA E FORTE
 CHE NEL PENSIER RINOVA LA PAURA!
 TANT'E' AMARA CHE POCO E' PIU' MORTE;
 MA PER TRATTAR DEL BEN CH'I' VI TROVAI,
 DIRO' DE L'ALTRE COSE CH'I' V'HO SCORTE.
 IO NON SO BEN RIDIR COM'I' V'INTRAI,
 TANT'ERA PIEN DI SONNO A QUEL PUNTO
 CHE LA VERACE VIA ABBANDONAI.

- 6.12 Modificate l'applicazione dell'esercizio precedente in modo che scriva il proprio output su un file di testo.

- 6.13 Scrivete un'applicazione che legga un file di testo e indichi quante volte vi compare ciascuna lettera dell'alfabeto.

- 6.14 Il comando `wc` di UNIX (*word count*) conta il numero di righe, di parole e di caratteri presenti in un file di testo. Costruite un programma che abbia le stesse funzionalità.

- 6.15 Costruite un'applicazione che controlli se in un sorgente Java le parentesi graffe aperte sono chiuse correttamente (si noti che *non* è sufficiente controllare che il numero di parentesi graffe aperte sia uguale al numero di parentesi graffe chiuse).

- 6.16 Utilizzando la classe `prog.utili.Sequenza` realizzate un'applicazione che legga una sequenza di frazioni da un file in cui le frazioni siano specificate una per riga nella forma numeratore/denominatore, e individui la frazione più vicina alla media.

- 6.17 Estendete l'applicazione dell'esercizio precedente in modo che, richiesta all'utente una coppia di frazioni f_1 e f_2 , visualizzi le frazioni della sequenza comprese nell'intervallo chiuso $[f_1, f_2]$.

6.18 Co
fig
re
qu
ce
spe
rag
leg
da

6.11

Sviluppo
consiste
nel file,

Per s
occorre
ge prog
precisan
E con un
è utile p
Occorre

• pu
C
sp

La classe

- pu
- Ind
- pu
- Re
- pu
- Re
- pu
- Re
- pu
- Re
- a c
- in
- Si
- co

- 6.18 Considerate file di testo contenenti descrizioni di figure: ciascuna riga specifica il tipo di figura e i dati che la definiscono. Ad esempio il file

```
rettangolo 4.2, 6
quadrato 3.1
cerchio 3.9
```

specificava un rettangolo di base 4.2 e altezza 6, un quadrato di lato 3.1 e un cerchio di raggio 3.9. Utilizzando la classe `prog.utili.Sequenza` scrivete un'applicazione che legga una sequenza di figure a partire da un file, calcoli la media delle aree e visualizzi i dati della figura la cui area è più vicina alla media.

6.11 Esempio: la tavola delle occorrenze

Svilupperemo ora un'altra applicazione, che chiameremo `OccorrenzeParole`, il cui compito consiste nel leggere un file di testo e produrre in output un elenco delle parole che compaiono nel file, indicando, per ognuna, quante volte è presente.

Per sviluppare l'applicazione avremo bisogno di tener conto, per prima cosa, del numero di occorrenze di una parola. A questo scopo utilizzeremo la classe generica `Occorrenza` del package `prog.utili`. Questa classe permette di rappresentare oggetti con associato un contatore. Più precisamente, la classe ha un tipo parametro `E`: un'istanza di `Occorrenza<E>` è un oggetto di tipo `E` con un contatore. Nell'esempio che stiamo sviluppando, un'istanza di `Occorrenza<String>` è utile per associare a una stringa il suo numero di occorrenze nel testo. La classe generica `Occorrenza<E>` mette a disposizione il seguente costruttore:

- `public Occorrenza(E e)`

Crea una nuova istanza della classe che permette di contare le occorrenze dell'oggetto specificato come argomento. Il valore iniziale del contatore è 1.

La classe mette inoltre a disposizione, fra gli altri, i seguenti metodi:

- `public void incrementa()`

Incrementa il contatore delle occorrenze dell'istanza che esegue il metodo.

- `public int getValoreContatore()`

Restituisce il valore del contatore delle occorrenze dell'istanza che esegue il metodo.

- `public E get()`

Restituisce un riferimento all'oggetto che viene contato dall'istanza che esegue il metodo.

- `public boolean equals(Occorrenza<E> altra)`

Restituisce `true` se l'occorrenza che esegue il metodo si riferisce a un oggetto uguale a quello a cui fa riferimento l'occorrenza fornita come argomento, e restituisce `false` in caso contrario. Il criterio di uguaglianza è quello definito dal metodo `equals` di `E`. Si osservi che due occorrenze di due oggetti uguali sono considerate uguali, anche se i contatori sono differenti.

- `public String toString()`

Restituisce una stringa che descrive l'occorrenza che esegue il metodo.

Per ogni parola che compare nel testo, costruiremo un oggetto di tipo `OcCorrenza<String>` al fine di contare le occorrenze. Avremo inoltre bisogno di realizzare la tavola delle occorrenze, cioè una struttura dati in cui memorizzare questi oggetti. Prima di scegliere la struttura dati possiamo comunque descrivere ad alto livello lo schema dell'applicazione:

```

while (ci sono righe nel file di testo) {
    preleva una riga dal file

    while (la riga contiene parole) {
        preleva una parola dalla riga
        if (la parola compare già nella tavola delle occorrenze)
            incrementa il contatore dell'occorrenza corrispondente
        else
            aggiungi la parola alla tavola delle occorrenze
    }
    stampa la tavola delle occorrenze
}

```

Per passare all'implementazione dobbiamo scegliere la struttura dati con cui rappresentare la tavola delle occorrenze. Gli array non sono, in questo caso, una struttura adatta, perché non siamo in grado di conoscere a priori il numero di parole che possiamo trovare in un file di testo. In questa circostanza risulta più adatta una struttura dati in cui si possano aggiungere liberamente nuovi elementi. Utilizzeremo a questo scopo la classe generica `Sequenza` del package `prog.util`, presentata nel Paragrafo 5.7. Ricordiamo brevemente che un'istanza di `Sequenza<E>` permette di modellare una sequenza di oggetti di tipo E. Per il problema in esame, gli oggetti della sequenza saranno occorrenze di stringhe. In altre parole, utilizzeremo la classe `Sequenza` con tipo argomento `OcCorrenza<String>`.

La parte del codice che si occupa di estrarre le righe dal file e le parole da una riga può essere facilmente implementata utilizzando la classe `FileInputManager` del package `prog.io` e la classe `StringTokenizer` del package `java.util` descritta nel Paragrafo 4.9. A questo scopo assumiamo che `fin` sia un riferimento all'oggetto di tipo `FileInputManager` che realizza il canale di comunicazione con il file di testo.

```

String riga;
//fintantoché il file contiene delle righe, legge una riga dal file...
while ((riga = fin.readLine()) != null) {
    StringTokenizer stk = new StringTokenizer(riga, " \t,.;:'?!");
    //...fintantoché la riga contiene altre parole...
    while (stk.hasMoreTokens()) {
        //...preleva una parola dalla riga...
        String parola = stk.nextToken().toLowerCase();
    }
}

```

```

if (la parola compare già nella tavola delle occorrenze)
    incrementa il contatore dell'occorrenza corrispondente
else
    aggiungi la parola alla tavola delle occorrenze
}
stampa la tavola delle occorrenze

```

Si osservi che la stringa " \t,.,;:'?!", fornita come argomento al costruttore della classe `StringTokenizer`, specifica che ognuno dei caratteri indicati, cioè lo spazio, il carattere di tabulazione ('\t') e i segni di interpunkzione, deve essere considerato dall'estrattore di token come separatore.

Per quel che riguarda la parte di gestione della tavola delle occorrenze, prima del ciclo esterno dovremo creare la struttura dati che utilizzeremo per memorizzarla tramite l'istruzione

```

Sequenza<Occorrenza<String>> tavolaOcorrenze =
    new Sequenza<Occorrenza<String>>();

```

Si noti, nell'istruzione precedente, l'indicazione del tipo argomento `Occorrenza<String>`, sia nella dichiarazione di `tavolaOcorrenze` che nella chiamata del costruttore. In entrambi casi, per la classe generica `Occorrenza` viene indicato il tipo argomento `String`. In altre parole, stiamo considerando una sequenza formata da occorrenze di stringhe. Passiamo ora a implementare lo schema

```

if (la parola compare già nella tavola delle occorrenze)
    incrementa il contatore dell'occorrenza corrispondente
else
    aggiungi la parola alla tavola delle occorrenze

```

Anzitutto dobbiamo trovare il modo di verificare se `parola` compare già nella tavola delle occorrenze. A questo scopo possiamo utilizzare il metodo `find` della classe `Sequenza`. Questo metodo riceve un riferimento `o` del tipo `E` e, utilizzando il metodo `equals` definito per l'oggetto riferito da `o`, cerca un elemento `e` nella sequenza tale che l'espressione `o.equals(e)` risulti vera. Si osservi che, in questo caso, la tavola delle occorrenze, cioè l'oggetto di tipo `Sequenza` cui chiediamo di eseguire il metodo `find`, contiene oggetti di tipo `Occorrenza<String>`. Pertanto l'oggetto da cercare, cioè l'argomento di `find`, non potrà essere `parola`, ma dovrà essere di tipo `Occorrenza<String>`. A tale scopo creiamo un'istanza di `Occorrenza<String>` per `parola` e la utilizziamo come argomento del metodo `find` nel modo seguente:

```

Occorrenza<String> nuovaOcc = new Occorrenza<String>(parola);
Occorrenza<String> occTrovata = tavolaOcorrenze.find(nuovaOcc);

```

Se l'elemento cercato esiste, il metodo `find` restituisce il riferimento a esso. Se invece la sequenza non lo contiene, il metodo restituisce `null`. Poiché il metodo `find` stabilisce l'uguaglianza in base al metodo `equals` degli oggetti memorizzati nella sequenza, in questo caso di tipo `Occorrenza<String>`, e il metodo `equals` di `Occorrenza`, come indicato sopra, stabilisce

l'uguaglianza in base a quella degli oggetti (in questo caso le stringhe) rappresentati, indipendentemente dai valori dei contatori, possiamo concludere che, nel caso la stringa rappresentata da `parola` sia già stata incontrata, e dunque vi sia un oggetto `Occorrenza<String>` nella sequenza per quella stringa, il metodo `find` restituisce un riferimento all'oggetto di cui dovremo incrementare il contatore. Se invece la stringa riferita da `parola` non è già stata incontrata in precedenza, occorrerà inserire un nuovo oggetto per tale stringa nella sequenza:

```
if (occTrovata != null)
    occTrovata.incrementa();
else
    tavolaOcorrenze.add(nuovaOcc);
```

Per concludere l'implementazione dell'applicazione dobbiamo visualizzare la tavola delle occorrenze. In questo caso utilizziamo un ciclo `for-each` che scandisce uno a uno gli elementi della sequenza:

```
for (Occorrenza<String> o : tavolaOcorrenze)
    out.println(o.toString());
```

Riportiamo infine il codice completo dell'applicazione. Si noti che il nome del file da analizzare dev'essere fornito come argomento sulla linea di comando.

```
import prog.io.*;
import prog.utili.Sequenza;
import prog.utili.Occorrenza;

import java.util.StringTokenizer;

class OccorrenzeParole {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleOutputManager out = new ConsoleOutputManager();
        FileInputStreamManager fin = new FileInputStreamManager(args[0]);

        //costruzione della struttura per contenere la tavola
        //delle occorrenze
        Sequenza<Occorrenza<String>> tavolaOcorrenze =
            new Sequenza<Occorrenza<String>>();

        String riga;
        //fintantoché il file contiene delle righe,
        //legge una riga dal file...
        while ((riga = fin.readLine()) != null){
```

Esercizi

6.19 Soluz.

ca

nu

Ne

do

n,

e,

l,

m,

z,

```

 StringTokenizer stk = new StringTokenizer(riga, " \t,.;:?!\"'");

 //...fintantoché la riga contiene altre parole...
 while (stk.hasMoreTokens()) {
     //...ne estrae una...
     String parola = stk.nextToken().toLowerCase();

     //...inserisce la parola in una occorrenza...
     Occorrenza<String> nuovaOcc = new Occorrenza<String>(parola);
     //...cerca l'occorrenza nella tavola delle occorrenze...
     Occorrenza<String> occTrovata = tavolaOcorrenze.find(nuovaOcc);
     //...se la trova...
     if (occTrovata != null) //...incrementa il contatore
         occTrovata.incrementa();
     else
         //...altrimenti la aggiunge
         tavolaOcorrenze.add(nuovaOcc);

 }

}

//visualizza la lista delle occorrenze
out.println("Ocorrenze delle parole nel testo: ");
for (Occorrenza<String> o : tavolaOcorrenze)
    out.println(o.toString());
}
}

```

Esercizi

- 6.19 Scrivete un'applicazione di nome **OcorrenzeLettere** che legga un file di testo e produca in output un elenco delle lettere che compaiono nel file, ognuna corredata del proprio numero di presenze. Ad esempio, l'output prodotto sul file contenente il testo

Nel mezzo del cammin di nostra vita

dovrà essere:

n,3
e,3
l,2
m,3
z,2

```

o,2
d,2
c,1
a,3
i,3
s,1
t,2
r,1
v,1

```

- 6.20 Estendete l'applicazione dell'Esercizio 6.18 in modo che, scorrendo la sequenza che contiene le figure lette dal file, costruisca e visualizzi la tavola delle occorrenze delle figure che compaiono nella sequenza.

Ad esempio, eseguendo l'applicazione utilizzando il seguente file di input

```

rettangolo 3.2, 5
quadrato 6.3
cerchio 8
rettangolo 3.2, 5
cerchio 8

```

L'output prodotto potrebbe essere il seguente:

Occorrenze delle figure:

```

2 occorrenze di Rettangolo base = 3.2, altezza = 5.0
1 occorrenza di Quadrato lato = 6.3
2 occorrenze di Cerchio raggio = 8.0

```

Si osservi che in questo caso è necessario memorizzare nella tavola delle occorrenze oggetti di tipo `Occorrenza<Figura>` in modo che un'occorrenza possa riferirsi indifferentemente a un rettangolo, un cerchio o un quadrato. Di conseguenza il tipo della `Sequenza` preposta a contenere la tavola delle occorrenze dovrà essere `Sequenza<Occorrenza<Figura>>`.

6.12 Le interfacce Java

Con il termine “interfaccia” abbiamo indicato finora l’insieme dei messaggi cui un oggetto è in grado di rispondere. Il linguaggio Java mette a disposizione la parola chiave `interface` che, sebbene si traduca esattamente con interfaccia, modella un concetto diverso da quello attribuito al termine finora. In questo paragrafo introduciamo, tramite alcuni esempi, le interfacce Java.

Nel Paragrafo 5.7, presentando le classi generiche `Sequenza` e `SequenzaOrdinata`, avevamo osservato che, mentre è possibile costruire sequenze di oggetti di un qualunque tipo, la

costruzione in sequenza. Altrimenti, un tipo per altro, definendo questa in questione di uguali oggetti. Pertanto, Retta e Linea sono tipi diversi.

Sequenza è il comune tipo.

Il messaggio più avanzato corrisponde all'ordinamento.

D'altra parte, i relativi relativi propriamente di". C'è una parte.

Sequenza, Sequenza, Sequenza.

A questo punto c'è l'interfaccia non quella di frazioni. Le frazioni non ne sono.

5 Posizioni.

costruzione di sequenze ordinate richiede che tra gli oggetti del tipo considerato esista una relazione di ordine totale. In particolare, il metodo `add` della classe `SequenzaOrdinata` dev'essere in grado di stabilire la posizione corretta in cui inserire un elemento, al fine di mantenere la sequenza ordinata.

Abbiamo già osservato che non per tutti i tipi è definibile un ordine totale: ad esempio, per un tipo `Soprammobile` non appare evidente alcuna nozione di ordine totale tra i suoi oggetti. Per altri tipi la situazione è più problematica. Ad esempio, per la classe `Rettangolo` potremmo definire una relazione tra gli oggetti sulla base della relazione d'ordine tra le loro aree. Tuttavia, questa relazione identifica oggetti differenti: rettangoli che hanno la stessa area sono considerati in questa relazione "uguali".⁵ In sostanza, la relazione ottenuta non è compatibile con il concetto di uguaglianza definito dal metodo `equals` della classe `Rettangolo`. Più in generale, per gli oggetti di tipo `Figura` non siamo in grado di definire una ragionevole relazione d'ordine totale. Pertanto, non ci aspettiamo di potere utilizzare invocazioni di `SequenzaOrdinata` che abbiano `Rettangolo` o `Figura` come tipo argomento.

In effetti, scrivendo in un'applicazione il tipo parametrizzato

```
SequenzaOrdinata<Rettangolo>
```

il compilatore segnalerà il seguente messaggio d'errore:

```
type parameter prog.utili.Rettangolo is not within its bound
```

Il messaggio d'errore indica che il tipo `Rettangolo` specificato non rispetta un vincolo. Vedremo più avanti, nel dettaglio, come è espresso questo vincolo. Possiamo però anticipare che il vincolo corrisponde proprio alla richiesta che il tipo argomento della classe abbia la proprietà di essere ordinabile.

D'altra parte, per altre classi come ad esempio `String`, `Frazione` e `Integer` esistono naturali relazioni d'ordine tra le loro istanze: nel caso di `String` si tratta dell'ordine alfabetico, più propriamente dell'ordine lessicografico, negli altri due casi si tratta dell'usuale relazione "minore di". Come abbiamo mostrato, possiamo costruire sequenze ordinate di oggetti di queste classi. In particolare, possiamo scrivere i tipi parametrizzati

```
SequenzaOrdinata<String>
SequenzaOrdinata<Frazione>
SequenzaOrdinata<Integer>
```

A questo punto dovremmo chiederci come fa il compilatore a sapere che i tipi `String`, `Frazione` e `Integer` possiedono una relazione di ordine totale, e quindi ad accettare queste tre scritture, e non quella con il tipo `Rettangolo`. Quest'informazione è fornita dal fatto che le classi `String`, `Frazione` e `Integer` implementano l'*interfaccia Comparable*, del package `java.lang`.

Le interfacce Java permettono di specificare un insieme di *comportamenti* senza però fornirne l'implementazione. Esse specificano cioè il prototipo di alcuni metodi e il loro contratto, ma non ne forniscono l'implementazione, rimandandola alle classi che *implementano l'interfaccia*.

⁵ Possiamo addirittura osservare che per ogni rettangolo ve ne sono infiniti altri che hanno la stessa area.

I metodi specificati in un’interfaccia sono a tutti gli effetti metodi astratti e, come nel caso delle classi astratte, non è possibile costruire istanze di un’interfaccia. Tuttavia il nome di un’interfaccia definisce un tipo riferimento che, come vedremo, si colloca nella gerarchia dei tipi e può essere utilizzato per riferirsi a oggetti di classi che implementano l’interfaccia. In altre parole, a una variabile riferimento di tipo I, dove I è un’interfaccia, si possono assegnare riferimenti a oggetti che siano istanze di classi che implementano l’interfaccia.

L’interfaccia `Comparable` specifica un unico metodo il cui scopo è quello di definire un ordine totale sugli oggetti che implementano l’interfaccia. L’interfaccia è generica e prevede un tipo parametro T. Il metodo è così descritto:

- `public int compareTo(T o)`

Confronta l’oggetto che esegue il metodo con quello specificato come argomento, e restituisce un intero negativo, zero, o un intero positivo, a seconda che l’oggetto che esegue il metodo sia minore, uguale o maggiore di quello specificato come argomento.

Affinché una classe implementi un’interfaccia, è necessario anzitutto indicare il nome dell’interfaccia implementata nell’intestazione, utilizzando la parola chiave `implements`. Ad esempio l’intestazione della classe `Frazione` del package `prog.utili`, che implementa l’interfaccia `Comparable`, è:

```
public class Frazione implements Comparable<Frazione>
```

Si noti che, essendo `Comparable` un’interfaccia generica, viene fornito un tipo argomento, in questo caso `Frazione`. In altre parole, il metodo `compareTo` previsto dall’interfaccia avrà un parametro di tipo `Frazione`, cioè permetterà di confrontare una frazione, quella che esegue il metodo, con un’altra, quella fornita tramite l’argomento.

Una classe che implementa un’interfaccia si impegna a fornire l’implementazione di tutti i metodi descritti nell’interfaccia (in caso contrario la classe sarà astratta). Ad esempio la classe `Frazione` implementa il metodo indicato sopra, previsto dall’interfaccia `Comparable`, che permette di confrontare la frazione che esegue il metodo con quella fornita tramite l’argomento, definendo così una relazione d’ordine totale tra le frazioni. Fra le classi dei package di Java che abbiamo utilizzato finora, diverse implementano l’interfaccia `Comparable`: la classe `String` implementa `Comparable<String>` (in questo caso `compareTo` realizza l’ordinamento lessicografico), la classe involucro `Integer` implementa `Comparable<Integer>`.

Riassumendo, possiamo immaginare un’interfaccia come una *promessa*: un’interfaccia promette certi comportamenti. Una classe che implementa l’interfaccia soddisfa la promessa, cioè fornisce effettivamente tali comportamenti (eventualmente insieme ad altri). La stessa interfaccia può essere implementata da classi totalmente differenti, che non sono in relazione diretta nella gerarchia.

Tornando all’esempio iniziale, la classe `SequenzaOrdinata` richiede che il tipo argomento implementi l’interfaccia `Comparable`. In particolare, il metodo `compareTo` viene utilizzato nell’implementazione del metodo `add` per confrontare l’elemento da inserire con quelli già presenti nella sequenza, al fine di individuare la posizione corretta. Si osservi che la scelta della posizione in cui inserire il nuovo elemento non dipende dal particolare tipo di oggetti considerati,

ma
con
ti C
ord

inte
mer
UM
UM

nell
da u
pler
clas
nota
lizz
sim
zior
Fra
un C

6.1

Le c
dent
stan

6.
interf

ma unicamente dalla proprietà, descritta dall'interfaccia Comparable, di possedere un metodo con il quale stabilire l'ordine tra gli elementi. In altre parole, il fatto che una classe implementi Comparable è sufficiente a garantire che gli oggetti possano essere posti in una sequenza ordinata, *indipendentemente* dalla natura di tali oggetti.

Concludiamo questo paragrafo introducendo la notazione UML per le interfacce. In UML le interfacce sono rappresentate come le classi, ma nel comparto con il nome è indicato esplicitamente che si tratta di un'interfaccia tramite lo *stereotipo* <>interface>. Nella terminologia di UML uno stereotipo indica una specializzazione di un concetto del linguaggio. Un'interfaccia in UML non è altro che una classe che non fornisce l'implementazione dei metodi.

Ad esempio, l'interfaccia parametrizzata Comparable<Frazione> è rappresentata come nella Figura 6.14.⁶ La relazione fra un'interfaccia e una classe che la implementa è indicata da una *relazione di realizzazione*, ossia da una freccia tratteggiata che va dalla classe che implementa l'interfaccia all'interfaccia stessa. Ad esempio, nella Figura 6.14 è indicato che la classe Frazione implementa Comparable<Frazione>. Per evidenziare la differenza fra le notazioni, abbiamo rappresentato nella figura anche la classe Object e la relazione di generalizzazione che indica che Frazione eredita da Object. Non è un caso che le notazioni siano simili; infatti, sia la relazione di generalizzazione sia quella di realizzazione inducono una relazione supertipo/sottotipo. La relazione di generalizzazione mette in evidenza che ogni istanza di Frazione è un Object, mentre quella di realizzazione mostra che ogni istanza di Frazione è un Comparable.

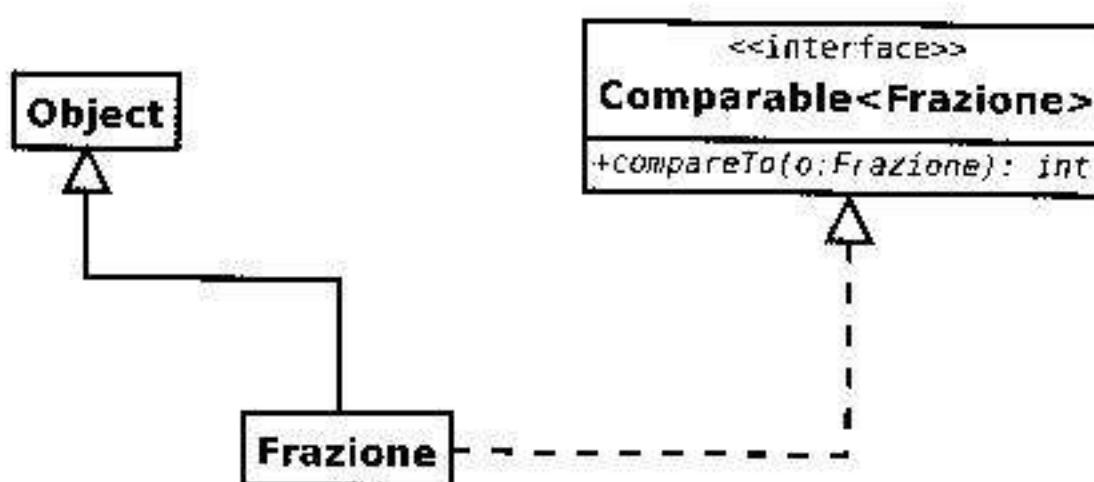


Figura 6.14 UML: rappresentazione di un'interfaccia.

6.13 L'interfaccia Iterable e il ciclo for-each

Le classi Sequenza, SequenzaOrdinata e Insieme che abbiamo utilizzato nei capitoli precedenti permettono di rappresentare collezioni di dati. Per queste classi, e per altre nella libreria standard di Java, è possibile scandire uno alla volta gli elementi di una collezione mediante l'uso

⁶ Qui utilizziamo una semplificazione della notazione UML, che prevede una rappresentazione apposita per classi e interfacce generiche mediante i cosiddetti *tipi parametrizzati*.

del ciclo `for-each`. Questa proprietà è garantita dal fatto che tutte queste classi implementano l’interfaccia `Iterable` definita nel package `java.lang`.

Ogni classe che implementa `Iterable` rappresenta una collezione di dati che può essere “iterata”, cioè scandita, un elemento alla volta, mediante l’uso di un oggetto che viene detto *iteratore*. Un iteratore, in sostanza, non è altro che un elenco degli elementi presenti nella collezione. Un iteratore, per una collezione di elementi di un tipo `E`, è un oggetto di tipo `Iterator<E>`. L’interfaccia generica `Iterable<E>` prevede il seguente metodo:

- `public Iterator<E> iterator()`

Restituisce un iteratore degli oggetti presenti nella collezione che esegue il metodo.

Ad esempio, ogni oggetto di tipo `Sequenza<Frazione>` dispone di un metodo `iterator` che restituisce un oggetto di tipo `Iterator<Frazione>`, che rappresenta un elenco di tutte le frazioni presenti nella sequenza.⁷ Ora studieremo più in dettaglio questo meccanismo, a partire dal quale è definito il ciclo `for-each` che abbiamo già utilizzato più volte per scandire delle collezioni di oggetti. Come abbiamo visto il metodo `iterator` restituisce un iteratore che fornisce un elenco di tutti gli elementi della collezione considerata. È possibile scandire gli elementi contenuti in tale elenco utilizzando i metodi:⁸

- `public E next()`

Restituisce il prossimo elemento dell’iteratore, eliminandolo dall’iteratore (non dalla struttura per la quale l’iteratore è stato costruito). Se l’iteratore è vuoto, si verifica un errore in fase di esecuzione.⁹

- `public boolean hasNext()`

Restituisce `true` se l’iteratore contiene degli elementi e `false` in caso contrario.

Un iteratore può essere dunque scandito utilizzando i due metodi `next` e `hasNext`: con quest’ultimo si controlla di non avere raggiunto la fine dell’iteratore, con il primo si può ottenere l’elemento successivo.

Consideriamo, ad esempio, l’applicazione `PappagalloFrazioni` presentata nel Paragrafo 5.7. Nella prima parte, viene letta una sequenza di frazioni e memorizzata in un oggetto di tipo `Sequenza<Frazione>`, accessibile tramite un riferimento di nome `frazioni`.

Possiamo costruire un iteratore per la sequenza scrivendo:

```
Iterator<Frazione> elenco = frazioni.iterator();
```

Quindi possiamo utilizzare l’iteratore riferito da `elenco` per scandire e visualizzare la sequenza mediante un ciclo `while` come il seguente:

⁷ I contratti dei metodi `iterator` forniti dalle classi `Sequenza` e `SequenzaOrdinata` prevedono inoltre che l’ordine degli elementi nell’iteratore corrisponda a quello della collezione di partenza. Nel caso della classe `Insieme` non vi è invece questa richiesta aggiuntiva, in quanto i suoi elementi non compaiono in un ordine particolare.

⁸ È previsto anche un metodo denominato `remove` che non utilizzeremo negli esempi seguenti.

⁹ Il contratto specificato nella documentazione di `Iterator` è più preciso di quello che abbiamo fornito noi. Descriveremo il contratto completo del metodo nel Capitolo 11, dopo avere introdotto le eccezioni.

```
while (elenco.hasNext())
    out.println(elenco.next());
```

In alternativa, possiamo scandire la sequenza con un ciclo `for` come il seguente:

```
for (Iterator<Frazione> elenco = frazioni.iterator(); elenco.hasNext();)
    out.println(elenco.next());
```

Il seguente ciclo `for-each`, più semplice e compatto, che abbiamo utilizzato nell'applicazione `PappagalloFrazioni`, produce lo stesso effetto e viene proprio realizzato ricorrendo, implicitamente, a un iteratore:

```
for (Frazione f : frazioni)
    out.println(f);
```

Se la classe `Sequenza` non implementasse l'interfaccia `Iterable` non sarebbe possibile scrivere questo ciclo.

Concludiamo questo paragrafo, osservando che abbiamo sempre parlato di "tipo" `Iterator` e non di "classe" `Iterator`. Ciò è dovuto al fatto che `Iterator` *non* è una classe, ma un'interfaccia definita nel package `java.util`. Dal punto di vista dell'uso, ciò è del tutto irrilevante: di fatto abbiamo utilizzato solo il tipo riferimento e i metodi promessi dall'interfaccia.

Esercizi

- 6.21 Modificate la classe `PappagalloFrazioni` in modo che visualizzi prima tutte le frazioni che si trovano in una posizione dispari della sequenza, poi tutte quelle che si trovano in una posizione pari. Ad esempio, se sono state inserite, nell'ordine, le frazioni $1/3$, $3/4$, $1/7$, $4/5$ e $3/8$, l'applicazione dovrà visualizzare prima l'elenco $1/3$, $1/7$, $3/8$ e poi l'elenco $3/4$, $4/5$. Sviluppate la soluzione in due modi: ricorrendo a cicli `for-each` (potete tener conto della parità della posizione negando, a ogni iterazione, una variabile di tipo `boolean`) e ricorrendo a un iteratore (a ogni iterazione potete chiamare due volte il metodo `next`, facendo attenzione alla fine dell'elenco).
- 6.22 Il package `prog.utili` fornisce una classe di nome `GestioneArray` che contiene alcuni metodi statici relativi agli array. Tra questi vi è un metodo `ordina` che riceve come argomento il riferimento a un array di oggetti e restituisce `void`. L'effetto del metodo è quello di ordinare gli elementi all'interno dell'array in modo crescente. È necessario fornire un array di elementi di un tipo che implementi l'interfaccia `Comparable`. Scrivete una semplice applicazione che legga 10 frazioni e le memorizzi in un array. L'applicazione dovrà poi ordinare l'array e visualizzare l'elenco delle frazioni ordinato.
- 6.23 Ripetete l'Esercizio 6.22 leggendo 10 stringhe anziché 10 frazioni.
- 6.24 Ripetete l'Esercizio 6.22 leggendo 10 `Integer` anziché 10 frazioni.

6.25 La libreria standard di Java, nella classe `Arrays` del package `java.util`, fornisce molti metodi per la gestione degli array:

- `public static void sort(Object[] a)`
Ordina l'array specificato come argomento in maniera crescente. Tutti gli elementi che compaiono nell'array devono essere istanze di classi che implementano l'interfaccia `Comparable`, altrimenti si verifica un errore in fase di esecuzione.
- `public static void sort(Object[] a, int fromIndex, int toIndex)`
Si comporta come il metodo precedente, ma anziché considerare l'intero array, considera e ordina solo la porzione che va dalla posizione specificata dal secondo argomento (compresa) fino a quella specificata dal terzo argomento (esclusa).

Ripetete gli Esercizi 6.22, 6.23 e 6.24 utilizzando questi metodi e senza fissare il numero esatto di dati da leggere, ma solo un numero massimo.

6.26 Ripetete l'Esercizio 6.25 considerando, come oggetti da ordinare, dei rettangoli. L'applicazione dovrebbe essere compilata correttamente, ma presentare problemi in esecuzione. Ripetete invece l'Esercizio 6.22 utilizzando rettangoli al posto di frazioni. In questo caso il compilatore non accetterà l'invocazione del metodo `ordina`. Mentre la segnatura del metodo `sort` della classe `Arrays` prevede un array di oggetti qualunque (il fatto che gli oggetti siano confrontabili, cioè implementino l'interfaccia `Comparable`, viene verificato solo in esecuzione, dal codice stesso del metodo), la segnatura del metodo `ordina` (che per il momento non indichiamo, in quanto piuttosto complicata) richiede invece che l'array fornito come argomento abbia un tipo base che implementa l'interfaccia `Comparable`. Pertanto, un array di tipo `Rettangolo` in questo caso non viene accettato in fase di compilazione.

6.14 La gerarchia dei tipi

Come abbiamo sottolineato nel Paragrafo 6.4, l'importanza dell'ereditarietà è legata alle relazioni che essa stabilisce tra i tipi riferimento. Oltre alle classi, i tipi riferimento corrispondono alle interfacce e agli array. Nel Capitolo 5 abbiamo utilizzato tipi riferimento ad array, mentre nel Paragrafo 6.13 abbiamo dichiarato alcune variabili di tipo riferimento, utilizzando i nomi delle interfacce `Iterator` e `Iterable`.¹⁰

Tutti i tipi riferimento si trovano all'interno della *gerarchia dei tipi riferimento*, in cima alla quale si trova il tipo riferimento `Object`, corrispondente appunto alla classe `Object`. Studieremo ora più in dettaglio questa gerarchia, riferendoci ai tipi corrispondenti alle classi e alle interfacce. Rimandiamo invece ai paragrafi successivi l'analisi di alcuni particolari aspetti legati ai tipi generici, insieme ad alcuni cenni relativi ai tipi array.

Osserviamo, prima di tutto, che la gerarchia dei tipi riferimento, ristretta ai soli tipi corrispondenti alle classi, coincide con la gerarchia delle classi stessa: come indicato del Paragrafo 6.6, se una classe `B` è sottoclasse di una classe `A`, allora il tipo riferimento `B` è sottotipo del tipo

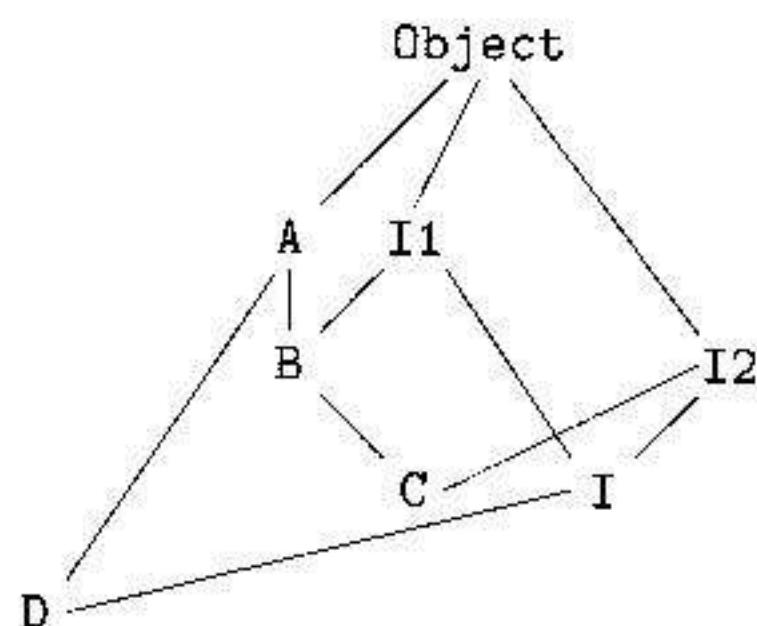
¹⁰ Trattandosi di due tipi generici, è stato fornito un tipo argomento.

riferimento A, e viceversa. Consideriamo ora un'interfaccia I1. Se la classe B implementa I1 allora il tipo riferimento I1 è un supertipo di B. Se C, a sua volta, è una classe che estende B, i tipi A, B e I1, oltre che C stesso, sono supertipi di C.¹¹

In generale, il nome di un'interfaccia è un supertipo di tutte le classi che implementano l'interfaccia.¹² Si noti che, mentre la superclasse diretta (cioè immediatamente al di sopra nella gerarchia) di una classe è unica, in quanto una classe può estendere solo un'altra classe, i supertipi diretti di un tipo riferimento possono essere più di uno. In particolare, una classe può implementare nessuna, una o più interfacce. Ad esempio, i supertipi che si trovano immediatamente al di sopra del tipo B sono I1 e A.

Supponiamo ora che vi sia un'altra interfaccia I2 e che essa sia implementata dalla classe C, ma non dalla classe B. Il tipo I2 è dunque un altro supertipo di C.

Un'interfaccia può essere definita estendendo una o più interfacce. Ciò significa che l'interfaccia eredita le "promesse" delle interfacce che estende. Il tipo riferimento così definito è un sottotipo dei tipi riferimento delle interfacce che vengono estese. Ad esempio, se l'interfaccia I estende I1 e I2, il tipo I è un sottotipo sia di I1 che di I2. Supponiamo che D sia una classe che implementa l'interfaccia I ed estende la classe A: i tipi Object, I1, I2, I e A sono supertipi di D (oltre che D stesso). La seguente figura riporta la parte di gerarchia dei tipi corrispondente alle classi e alle interfacce utilizzate come esempi in questa discussione (le linee congiungono dall'alto verso il basso supertipi con sottotipi diretti):



Possiamo effettuare assegnamenti tra tipi riferimento differenti, come già discusso nel Paragrafo 6.7 nel caso dei tipi riferimento corrispondenti alle classi. Ad esempio, un oggetto di tipo D può essere riferito da variabili di diversi tipi, tra cui A, I e I2. D'altra parte, a una variabile riferimento x di tipo I2, possono essere assegnati solo riferimenti di tipi I2, I, C e D (più eventuali loro sottotipi non considerati nell'esempio). Pertanto (limitandosi alle classi considerate nell'esempio), gli oggetti riferiti da x potranno essere solo istanze di C o D. Se I2 definisce un metodo astratto m privo di argomenti, la chiamata

`x.m();`

¹¹ Si suggerisce al lettore di seguire questa discussione disegnando le relazioni gerarchiche tra classi e interfacce, man mano che vengono menzionate.

¹² Comprese le loro sottoclassi. Ad esempio, la classe C implementa l'interfaccia I1 in virtù del fatto che I1 è implementata dalla superclasse B di C.

è lecita. Il metodo effettivamente chiamato dipenderà dal tipo dell'oggetto riferito da `x` al momento dell'esecuzione (tale metodo deve esistere perché gli oggetti riferibili da `x` appartengono a classi che sono sottotipi di `I2`).

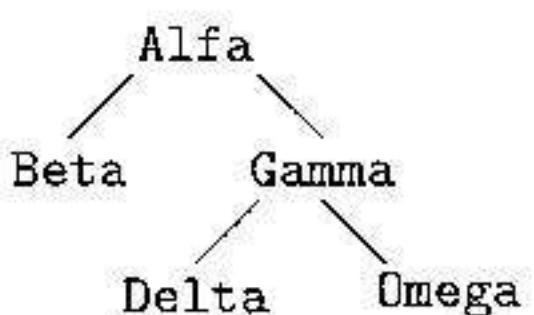
Concludiamo il paragrafo con una breve precisazione relativa alla terminologia per i tipi riferimento: diciamo che un tipo riferimento `S` estende un tipo riferimento `T` (`S extends T`), se e solo se `S` è un qualunque sottotipo di `T` (compreso `T` stesso). Questo equivale a dire che `T` è supertipo di `S` (indicato con `T super S`). Se sia `S` che `T` corrispondono a nomi di classi, questo significa che `S` è una sottoclasse di `T` (diretta o indiretta). Se `S` è una classe e `T` un'interfaccia, questo implica che `S` implementa (direttamente o indirettamente) l'interfaccia `T`. Si noti che non è possibile che `S` sia un'interfaccia e `T` una classe.

Esempio

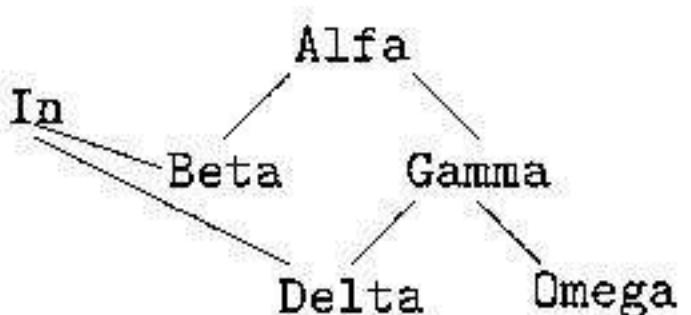
Consideriamo 4 classi `Alfa`, `Beta`, `Gamma` e `Delta` e un'interfaccia `In` tali che:

- `Beta` e `Gamma` estendano `Alfa`, `Delta` estenda `Gamma`, solo `Beta` e `Delta` implementino `In`
- la classe `Alfa` sia astratta, le altre classi siano concrete
- ognuna delle classi possiede un costruttore privo di argomenti
- in ognuna delle classi sia definito, tra gli altri, un metodo `public String toString()` che restituisca una stringa uguale al nome della classe stessa. Ad esempio, il metodo `toString` definito nella classe `Beta` restituisce la stringa "Beta".

La parte di gerarchia delle classi, contenente le classi considerate, è rappresentata nella seguente figura:



Considerando anche l'interfaccia `In`, si ottiene la seguente gerarchia dei tipi (in cui non è rappresentato il tipo `Object`, supertipo di tutti i tipi considerati):



Consideriamo ora le affermazioni sottoelencate:

1. In è un supertipo di Gamma
2. I metodi astratti di Alfa sono implementati in Gamma
3. Delta è una sottoclassc di Beta
4. In è un supertipo di Alfa
5. Object è un supertipo di Gamma
6. In è un supertipo di Beta
7. Alfa è un tipo riferimento
8. Alfa è il nome di un costruttore
9. In è il nome di un costruttore
10. Delta è una sottoclasse di Alfa
11. Delta è una sottoclasse di Object
12. Beta è una sottoclasse di In
13. I metodi astratti di In sono implementati in Delta
14. In possiede un costruttore.

Osservando la gerarchia dei tipi e la gerarchia delle classi, si può dedurre immediatamente che le affermazioni 1, 3, 4 sono false, mentre le affermazioni 5, 6, 10, 11 sono vere. Analizziamo più in dettaglio le altre affermazioni.

La numero 2 (i metodi astratti di Alfa sono implementati in Gamma) è vera, in quanto una classe concreta che estende una classe astratta deve implementarne i metodi astratti.

La numero 7 (Alfa è un tipo riferimento) è vera, in quanto ogni nome di classe è anche il nome di un tipo riferimento.

La numero 8 (Alfa è il nome di un costruttore) è vera: ogni classe possiede un costruttore, anche se è astratta. Questo fatto, accennato nel Paragrafo 6.5, verrà approfondito nel Capitolo 8. Al contrario, le interfacce non definiscono oggetti e pertanto non possiedono costruttori. Pertanto, le affermazioni numero 9 e 14 (In è il nome di un costruttore, In possiede un costruttore) sono false.

La numero 12 (Beta è una sottoclasse di In) è falsa: In non è una classe, dunque Beta non può esserne una sottoclasse; il tipo riferimento Beta invece è un sottotipo del tipo riferimento In.

Infine, l'affermazione numero 13 (i metodi astratti di In sono implementati in Delta) è vera, in quanto una classe che implementa un'interfaccia deve fornire l'implementazione di tutti i metodi previsti dall'interfaccia.

Consideriamo ora le seguenti dichiarazioni di variabile:

```

Alfa p;          6. q
Gamma q;        7. p
Beta t;         8. r
Delta r;        9. p
In s;

```

e i seguenti assegnamenti:

1. s = new In() 10. s
2. s = new Beta() 11. s
3. p = new Alfa() 12. t
4. t = (Beta) t 13. r
5. t = (Alfa) t 14. p
6. q = (Gamma) r Conclu...
public
Gamm...
Alfa...
In y...
Syst...
Syst...
Alfa...
Syst...
}
7. p = new Delta() 15. r
8. r = (Delta) q 16. r
9. p = t 17. r
10. s = r 18. r
11. s = q 19. r
12. t = p 20. r
13. r = q 21. r
14. p = (Beta) s 22. r

Analizziamo gli assegnamenti dal punto di vista del compilatore: alcuni di essi saranno compilati (eventualmente con promozioni di tipo), mentre altri no, per diverse ragioni. Esaminiamoli uno per uno:

1. s = new In(): non compilato (le interfacce non possiedono costruttori)
2. s = new Beta(): compilato (promozione implicita)
3. p = new Alfa(): non compilato (Alfa è astratta, dunque non può essere istanziata)
4. t = (Beta) t: compilato
5. t = (Alfa) t: non compilato (il lato destro è di tipo Alfa, mentre quello sinistro è del sottotipo Beta)

6.15

Come trovare il tipo di un'espressione all'interno di un'interfaccia come esempio di tipo generico. Prima si definisce un tipo generico e si specifica le dichiarazioni.

6. `q = (Gamma) r;` compilato
7. `p = new Delta();` compilato (promozione implicita)
8. `x = (Delta) q;` compilato
9. `p = t;` compilato (promozione implicita)
10. `s = r;` compilato (promozione implicita)
11. `s = q;` non compilato (il tipo del lato sinistro non è un supertipo del tipo del lato destro)
12. `t = p;` non compilato
13. `x = q;` non compilato
14. `p = (Beta) s;` compilato (promozione implicita da Beta a Alfa)

Concludiamo questo esempio, considerando il seguente metodo `main`:

```
public static void main(String[] args) {
    Gamma w = new Delta();
    Alfa x = w;
    In y = new Beta();
    System.out.println(x.toString()); // Delta
    System.out.println(y.toString()); // Beta
    Alfa u = (Beta) y; // Errore
    System.out.println(u.toString()); // Errore
}
```

Si noti l'uso dei tipi riferimento: ad esempio si osservi l'uso del riferimento `y` di tipo `In`. Le varie chiamate dei metodi `toString` sono polimorfe: il metodo chiamato dipende dal tipo effettivo dell'oggetto e non da quello del riferimento. Pertanto, gli output prodotti saranno, nell'ordine, le stringhe "Delta", "Beta" e "Beta".

6.15 Gerarchia dei tipi e tipi generici

Come tutti i tipi riferimento, anche i tipi generici e i relativi tipi parametrizzati si collocano all'interno della gerarchia dei tipi. In questo paragrafo studiamo come ciò avvenga, utilizzando come esempio la classe `Sequenza` introdotta nel Paragrafo 5.7.

Prima di tutto diciamo che, per scelta dei progettisti del linguaggio, dato un tipo generico `Tipo<E>` e due tipi `A` e `B`, *non vi è alcuna relazione gerarchica tra i tipi parametrizzati `Tipo<A>` e `Tipo`, indipendentemente da eventuali relazioni gerarchiche tra i tipi argomenti `A` e `B`.* Ad esempio, `Sequenza<Rettangolo>` *non è un supertipo di `Sequenza<Quadrato>`.* Pertanto, date le dichiarazioni di variabili:

```
Sequenza<Rettangolo> sr;
Sequenza<Quadrato> sq;
```

entrambi i seguenti assegnamenti non sono leciti:

```
sq = sr;
sr = sq;
```

Mentre il fatto che il primo assegnamento non sia lecito è del tutto intuitivo, il fatto che non lo sia nemmeno il secondo è, a prima vista, sorprendente. Mostreremo ora quali problemi si potrebbero verificare durante l'esecuzione se Sequenza<Rettangolo> fosse supertipo di Sequenza<Quadrato> e, dunque, se il secondo assegnamento fosse lecito.

Consideriamo la seguente applicazione che legge una sequenza di quadrati e comunica la somma delle loro aree e la somma dei loro lati. Nella fase di lettura, la sequenza dei quadrati viene memorizzata in un oggetto di tipo Sequenza<Quadrato> riferito dalla variabile sq dello stesso tipo:

```
import prog.io.*;
import prog.utili.Sequenza;
import prog.utili.Quadrato;

class SequenzaQuadrati {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        Sequenza<Quadrato> sq = new Sequenza<Quadrato>();

        //fase di lettura
        double x = in.readDouble("Inserire il lato di un quadrato" +
            " (0 per terminare)");
        while (x != 0) {
            //crea un nuovo quadrato e lo aggiunge alla sequenza
            sq.add(new Quadrato(x));
            x = in.readDouble("Inserire il lato di un quadrato" +
                " (0 per terminare)");
        }

        //calcolo e comunicazione della somma delle aree
        double sommaAree = 0;
        for (Quadrato q : sq)
            sommaAree += q.getArea();
        out.println("La somma delle aree vale " + sommaAree);
    }
}
```

```
//  
do  
fo  
ou  
}  
}
```

Si supponga

Seque

Se ciò fo
letta. D'
del tipo p
il metod
parole, s

Seque
sr.ad

porterem
Sequenz
rendendo
questa ra
nerale Ti
non è su
disporre

Per o
convenzi
gomento
siamo leg
testo fun
è un caso
Pertanto,
e Sequer

¹³ Dunq
osservi che
new Sequen

Siccome un
¹⁴ Nella
utilizzato ai

```

//calcolo e comunicazione della somma dei lati
double sommaLati = 0;
for (Quadrato q : sq)
    sommaLati += q.getLato();
out.println("La somma dei lati vale " + sommaLati);
}
}

```

Si supponga di introdurre subito dopo il ciclo while l'istruzione

```
Sequenza<Rettangolo> sr = sq;
```

Se ciò fosse lecito, dopo l'esecuzione di quest'istruzione `sq` e `sr` si riferirebbero alla sequenza letta. D'altra parte, la classe `Sequenza<E>` dispone del metodo `add` che aggiunge un elemento del tipo parametro `E` alla sequenza. Utilizzando questo metodo, potremmo aggiungere, invocando il metodo `add` tramite il riferimento `sr`, un oggetto di tipo `Rettangolo` alla sequenza. In altre parole, se dopo il ciclo `while` scrivessimo le istruzioni:

```
Sequenza<Rettangolo> sr = sq; //questo non è permesso
sr.add(new Rettangolo(1, 2));
```

porteremmo la sequenza in uno stato inconsistente. In particolare, tramite la variabile `sq` di tipo `Sequenza<Quadrato>`, si accederebbe a una sequenza che contiene sia quadrati che rettangoli, rendendo impossibile l'esecuzione, nella parte rimanente del codice, del metodo `getLato`. Per questa ragione `Sequenza<Rettangolo>` non è supertipo di `Sequenza<Quadrato>` e, più in generale `Tipo<A>` non è mai supertipo di `Tipo`.¹³ Di conseguenza, anche `Sequenza<Object>` non è supertipo di `Sequenza<Quadrato>` e di `Sequenza<Rettangolo>`. Come facciamo a disporre di un tipo che possa essere utilizzato per riferirsi a sequenze di qualsiasi tipo?

Per ogni tipo generico, esiste un supertipo comune a tutti i suoi tipi parametrizzati, che per convenzione si denota utilizzando il simbolo speciale `?` (punto interrogativo) al posto del tipo argomento. Ad esempio, nel caso di `Sequenza` questo supertipo comune è `Sequenza<?>` (che possiamo leggere come “sequenza di tipo sconosciuto”). In altre parole, il simbolo `?` in questo contesto funge da *segnaposto* per un tipo non noto al momento della compilazione.¹⁴ Ogni sequenza è un caso particolare di `Sequenza<?>`, dove il segnaposto viene sostituito con un tipo effettivo. Pertanto, `Sequenza<?>` è supertipo, tra gli altri, di `Sequenza<Object>`, `Sequenza<Quadrato>` e `Sequenza<Rettangolo>`.

¹³ Dunque, nella gerarchia dei tipi di Java, una sequenza di quadrati non è una sequenza di rettangoli. Tuttavia si osservi che una sequenza di rettangoli può contenere dei quadrati. Se definiamo: `Sequenza<Rettangolo> seq = new Sequenza<Rettangolo>()`, il metodo `add` dell'oggetto riferito da `seq` riceve un argomento di tipo `Rettangolo`. Siccome un `Quadrato` è anche un `Rettangolo`, se `q` è di tipo `Quadrato` possiamo quindi scrivere `seq.add(q)`.

¹⁴ Nella terminologia tecnica in inglese, il simbolo `?` utilizzato come segnaposto viene detto *wildcard*, termine utilizzato anche per il jolly in un mazzo di carte.

Nell'applicazione precedente si potrebbe introdurre, dopo la lettura, l'assegnamento

```
Sequenza<?> s = sq;
```

Si osservi che l'invocazione

```
s.add(x)
```

non è permessa, *qualunque sia* il tipo di x. Infatti, il metodo add di un oggetto Sequenza di tipo E riceve come argomento un riferimento di tipo E (o di un suo sottotipo). Pertanto, affinché la chiamata sia lecita, il compilatore deve verificare che il tipo di x sia un sottotipo del tipo argomento corrispondente alla sequenza. Tale verifica in questo caso non è possibile, in quanto il metodo è invocato tramite il riferimento s, il cui tipo argomento per il compilatore è sconosciuto.¹⁵

Esempio

La seguente applicazione permette all'utente di inserire una sequenza di numeri con la virgola, oppure di numeri interi o, infine, di stringhe e la visualizza.

```
import prog.io.*;
import prog.utili.Sequenza;

class TestSequenze {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //scelta del tipo di sequenza da trattare
        out.println("Scelte disponibili:");
        out.println(" d = sequenza numeri con la virgola ");
        out.println(" i = sequenza numeri interi ");
        out.println(" s = sequenza stringhe ");
        char scelta = in.readChar("scelta (d|i|s)? ");
        while (scelta != 'd' && scelta != 'i' && scelta != 's') {
            out.println("scelta non valida: ripetere l'inserimento");
            scelta = in.readChar("scelta (d|i|s)? ");
        }

        //lettura e stampa della sequenza
        switch (scelta) {
```

¹⁵ L'invocazione s.add(null) è invece permessa in quanto il letterale null è assegnabile a *qualsiasi* tipo riferimento.

```

case 'd':
    //lettura sequenza di numeri double
    Sequenza<Double> sd = new Sequenza<Double>();
    double d;
    while ((d = in.readDouble("Inserire un numero reale" +
                               " (0 per terminare) ")) != 0)
        sd.add(d);
    //stampa
    for (Double d2 : sd)
        out.println(d2);
    break;

case 'i':
    //lettura sequenza di numeri interi
    Sequenza<Integer> si = new Sequenza<Integer>();
    int i;
    while ((i = in.readInt("Inserire un numero intero" +
                           " (0 per terminare) ")) != 0)
        si.add(i);
    //stampa
    for (Integer i2 : si)
        out.println(i2);
    break;

case 's':
    //lettura sequenza di stringhe
    Sequenza<String> ss = new Sequenza<String>();
    String s;
    while (!(s = in.readLine("Inserire una stringa" +
                             " (riga vuota per terminare) ")).equals(""))
        ss.add(s);
    //stampa
    for (String s2 : ss)
        out.println(s2);
    break;

}
}
}

```

Si osservi che, dopo la scelta dell'utente, l'applicazione è suddivisa in tre parti, ognuna delle quali si occupa della lettura e della stampa relativa al tipo di sequenza selezionato. In particolare,

alla fine di ognuna di esse, vi è un ciclo `for-each` che scandisce la sequenza e ne visualizza gli elementi.

Quest'azione è la stessa per tutti e tre i tipi di sequenze considerate: essa non dipende dal tipo di oggetti contenuti nella sequenza. È opportuno pertanto scrivere il ciclo una sola volta, e dedicare il costrutto `switch` alla parte che dipende dal tipo di sequenza considerata, cioè la lettura. Riscriviamo dunque l'applicazione suddividendola nelle tre fasi logiche che la costituiscono: la selezione del tipo di oggetti da trattare, la lettura della sequenza, la visualizzazione della sequenza. La fase di selezione non ha bisogno di alcuna modifica e, come prima, produce all'interno della variabile `scelta` di tipo `char`, un carattere che indica la scelta effettuata. La fase di lettura dovrà produrre all'interno di una variabile `seq` il riferimento alla sequenza letta. Pertanto, la fase finale di visualizzazione può essere scritta semplicemente come:

```
for (Object o : seq)
    out.println(o);
```

Sviluppiamo la fase di lettura cercando di utilizzare ciò che abbiamo già scritto. Se è stato scelto l'inserimento di numeri con la virgola, la sequenza sarà riferita da `sd` di tipo `Sequenza<Double>`, se è stato scelto l'inserimento di numeri interi la sequenza sarà accessibile tramite la variabile riferimento `si` di tipo `Sequenza<Integer>`. Infine, se è stato scelto l'inserimento di stringhe, la sequenza sarà accessibile da `ss` di tipo `Sequenza<String>`. Pertanto, alla fine della fase di lettura, occorre semplicemente assegnare alla variabile `seq` il riferimento tra `sd`, `si` e `ss` corrispondente al tipo in esame. In altre parole, dovremo essere in grado di eseguire uno dei tre assegnamenti `seq = sd`, `seq = si` e `seq = ss`. Pertanto il tipo di `seq` dev'essere un supertipo dei tipi di `sd`, `si` e `ss`, cioè dei tipi `Sequenza<Double>`, `Sequenza<Integer>` e `Sequenza<String>`. Come osservato in precedenza, `Sequenza<Object>` non è un supertipo di questi tipi. Al contrario, `Sequenza<?>` lo è. Pertanto dichiariamo `seq` di tipo `Sequenza<?>`.

Segue il codice dell'applicazione così modificata:

```
import prog.io.*;
import prog.utili.Sequenza;

class TestSequenze {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //scelta del tipo di sequenza da trattare
        out.println("Scelte disponibili:");
        out.println(" d = sequenza numeri con la virgola ");
        out.println(" i = sequenza numeri interi ");
        out.println(" s = sequenza stringhe ");
        char scelta = in.readChar("scelta (d|i|s)? ");
        while (scelta != 'd' && scelta != 'i' && scelta != 's') {
```

```
out.println("scelta non valida: ripetere l'inserimento");
scelta = in.readChar("scelta (d|i|s)? ");
}

//definizione variabile utilizzata, dopo la lettura, per
//riferirsi alla sequenza
Sequenza<?> seq = null;

//lettura della sequenza
switch (scelta) {
case 'd':
    Sequenza<Double> sd = new Sequenza<Double>();
    double d;
    while ((d = in.readDouble("Inserire un numero reale" +
                               " (0 per terminare) ")) != 0)
        sd.add(d);
    seq = sd;
    break;

case 'i':
    Sequenza<Integer> si = new Sequenza<Integer>();
    int i;
    while ((i = in.readInt("Inserire un numero intero" +
                               " (0 per terminare) ")) != 0)
        si.add(i);
    seq = si;
    break;

case 's':
    Sequenza<String> ss = new Sequenza<String>();
    String s;
    while (!(s = in.readLine("Inserire una stringa" +
                               " (riga vuota per terminare) ")).equals(""))
        ss.add(s);
    seq = ss;
    break;
}

//stampa della sequenza
for (Object o : seq)
    out.println(o);
```

```
}
```

```
}
```

Concludiamo il paragrafo aggiungendo un'ulteriore funzionalità all'applicazione `TestSequenze`: vogliamo che l'applicazione, dopo avere stampato la sequenza, indichi l'elemento centrale. Ad esempio, se la sequenza contiene cinque elementi, l'applicazione dovrà indicare il terzo elemento inserito. Nel caso di sequenza di lunghezza pari, l'applicazione dovrà indicare uno qualsiasi dei due elementi centrali.

A tale scopo, scorriamo gli elementi della sequenza, contandoli man mano, sino alla posizione centrale, calcolata sulla base della lunghezza della sequenza che è ottenibile invocando il metodo `size`. In particolare, utilizziamo il seguente schema, basato su un ciclo `for`:

```
Object centrale = primo elemento della sequenza
for (int i = 0; i < seq.size() / 2; i++)
    centrale = elemento successivo della sequenza
```

Si noti che, poiché il tipo degli oggetti di `seq` non è noto, la variabile `centrale` è dichiarata del tipo più ampio possibile, cioè `Object`.

Per scorrere gli elementi della sequenza, possiamo ricorrere a un iteratore, come descritto nel Paragrafo 6.13: invocando il metodo `iterator` di `seq` otteniamo un iteratore. A ogni invocazione del metodo `next` dell'iteratore otteniamo, uno dopo l'altro, gli elementi della sequenza. Come dichiarare l'iteratore? Nel Paragrafo 6.13 per scorrere una sequenza di oggetti di tipo `Frazione` abbiamo dichiarato un iteratore di tipo `Iterator<Frazione>`. In questo caso dobbiamo scorrere la sequenza `seq` il cui tipo non è noto, se non al momento dell'esecuzione: pertanto non è possibile conoscere nemmeno il tipo argomento dell'iteratore. Quindi, definiamo l'iteratore utilizzando il segnaposto `?` al posto del tipo argomento

```
Iterator<?> elenco = seq.iterator();
```

Se la sequenza è vuota non ha alcun senso determinarne l'elemento centrale. Dunque, questa parte verrà eseguita dopo avere controllato che la sequenza non sia vuota. Ecco la parte di codice che determina l'elemento centrale:¹⁶

```
if (!seq.isEmpty()) {
    //determina l'elemento centrale della sequenza
    Iterator<?> elenco = seq.iterator();
    Object centrale = elenco.next();
    for (int i = 0; i < seq.size() / 2; i++)
        centrale = elenco.next();
    out.println("Elemento centrale: " + centrale);
}
```

¹⁶ È necessario ricordarsi di inserire la direttiva di importazione per l'interfaccia `java.util.Iterator`.

Esercizi

6.27 Nell'esempio precedente, in realtà, si potrebbe iniziare costruendo una sequenza vuota di Object e aggiungere gli elementi ad essa. Tale soluzione risulterebbe però molto complicata se, per ciascun tipo selezionabile, si richiedesse l'esecuzione di operazioni specifiche per quel tipo. Ispirandovi a TestSequenze, scrivete un'applicazione che permetta all'utente di selezionare un tipo di figura (quadrato, rettangolo o cerchio). L'applicazione deve leggere una sequenza di figure del tipo selezionato. Nel caso dei quadrati, l'applicazione dovrà calcolare la media dei lati e la somma delle aree dei quadrati con lato superiore alla media. Nel caso dei cerchi, l'applicazione dovrà calcolare la media delle aree e indicare quanti cerchi hanno area inferiore alla media. Nel caso dei rettangoli, l'applicazione dovrà calcolare la media delle basi e la media delle altezze e dovrà indicare quanti rettangoli hanno la base minore della media delle basi e l'altezza maggiore della media delle altezze. Infine, l'applicazione dovrà elencare tutte le figure lette.

6.16 Vincoli sui segnaposto

Abbiamo mostrato che il carattere ? utilizzato al posto del tipo, come argomento di Sequenza, funge da segnaposto per un *qualsiasi* tipo riferimento, sconosciuto al momento della compilazione. È possibile limitare l'insieme dei tipi sostituibili al segnaposto a una sola parte della gerarchia, specificando, insieme al segnaposto, un vincolo, in uno dei seguenti modi:

- ? extends T
Il tipo sconosciuto può essere un qualunque sottotipo del tipo T indicato (compreso T stesso).
- ? super T
Il tipo sconosciuto può essere un qualunque supertipo del tipo T indicato (compreso T stesso).

Illustriamo, con un esempio, l'uso del primo tipo di vincolo. Scriviamo un'applicazione, che chieda all'utente di scegliere un tipo di figura, legga i dati di una sequenza di figure del tipo selezionato, e infine visualizzi la sequenza di figure lette. Possiamo scrivere facilmente l'applicazione, adattando TestSequenze. Considerando, per brevità, solo quadrati e cerchi, otteniamo la seguente applicazione:

```
import prog.io.*;
import prog.utili.Sequenza;
import prog.utili.Quadrato;
import prog.utili.Cerchio;

class TestSequenzeFigure {
```

```

public static void main(String[] args) {
    //predisposizione dei canali di comunicazione
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();

    //scelta del tipo di sequenza da trattare
    out.println("Scelte disponibili:");
    out.println(" c = sequenza di cerchi ");
    out.println(" q = sequenza di quadrati ");
    char scelta = in.readChar("scelta (c|q)? ");
    while (scelta != 'c' && scelta != 'q') {
        out.println("scelta non valida: ripetere l'inserimento");
        scelta = in.readChar("scelta (c|q)? ");
    }

    //definizione variabile utilizzata, dopo la lettura, per
    //riferirsi alla sequenza
    Sequenza<?> seq = null;

    //lettura della sequenza
    switch (scelta) {
        case 'c':
            Sequenza<Cerchio> sc = new Sequenza<Cerchio>();
            double r;
            while ((r = in.readDouble("Inserire la lunghezza del raggio" +
                " (0 per terminare) ")) != 0)
                sc.add(new Cerchio(r));
            seq = sc;
            break;

        case 'q':
            Sequenza<Quadrato> sq = new Sequenza<Quadrato>();
            int l;
            while ((l = in.readInt("Inserire la lunghezza del lato" +
                " (0 per terminare) ")) != 0)
                sq.add(new Quadrato(l));
            seq = sq;
            break;
    }

    //stampa della sequenza
    for (Object o : seq)

```

```

    out.println(o);

}
}

```

Dato che la sequenza letta è di figure, potremmo essere indotti a modificare il ciclo finale di stampa, nel seguente modo (dopo avere importato la classe **Figura**):

```

for (Figura f : seq)
    out.println(f);

```

Il compilatore segnalerà un errore sull'intestazione del ciclo **for**: la variabile **f** di tipo **Figura** non può essere utilizzata per scandire l'oggetto riferito da **seq**. In particolare, la variabile **seq** è stata dichiarata di tipo **Sequenza<?>**, cioè sequenza di un tipo qualunque, sconosciuto al momento della compilazione. Pertanto, nel caso più generale gli elementi della sequenza saranno **Object** e quindi non assegnabili a **Figura**. In questa situazione, poiché l'azione da eseguire nel ciclo è la stampa, l'utilizzo di una variabile di tipo **Object** per scandire la sequenza, permette di ottenere l'effetto desiderato.

Supponiamo, invece, di voler elencare le aree delle figure presenti nella sequenza. Nel ciclo finale possiamo scandire la sequenza di figure e per ognuna di esse invocare il metodo **getArea**, visualizzandone il risultato:

```

for (Object o : seq)
    out.println(o.getArea());

```

In questo caso il compilatore fornirà un'indicazione d'errore sull'invocazione del metodo **getArea**, non disponibile per il tipo **Object** del riferimento **o**. Un modo per risolvere questo problema, è introdurre una forzatura di **o** al tipo **Figura** prima di invocare il metodo **getArea**; tale forzatura non provocherà errori in esecuzione, poiché per la sua stessa costruzione, la sequenza contiene solo oggetti di tipo **Figura**. Utilizzando i tipi generici, possiamo tuttavia procedere in maniera più elegante e sicura: nella dichiarazione della variabile **seq** indichiamo esplicitamente che la sequenza potrà contenere esclusivamente oggetti di sottotipi del tipo **Figura**. A tale scopo, la dichiarazione verrà scritta esplicitando un vincolo sul segnaposto **? come**:

Sequenza<? extends Figura> seq

È ora possibile scrivere il ciclo **for**, utilizzando come indice una variabile di tipo **Figura** (il tipo più ampio che rispetta il vincolo indicato con il segnaposto):

```

for (Figura f : seq)
    out.println(f.getArea());

```

Ecco il testo dell'applicazione così riscritta:

```

import prog.io.*;
import prog.utili.Sequenza;

```

```

import prog.utili.Quadrato;
import prog.utili.Cerchio;
import prog.utili.Figura;

class TestSequenzeFigure {
    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //scelta del tipo di sequenza da trattare
        out.println("Scelte disponibili:");
        out.println(" c = sequenza di cerchi ");
        out.println(" q = sequenza di quadrati ");
        char scelta = in.readChar("scelta (c|q)? ");
        while (scelta != 'c' && scelta != 'q') {
            out.println("scelta non valida: ripetere l'inserimento");
            scelta = in.readChar("scelta (c|q)? ");
        }

        //definizione variabile utilizzata, dopo la lettura, per
        //riferirsi alla sequenza
        Sequenza<? extends Figura> seq = null;

        //lettura della sequenza
        switch (scelta) {
            case 'c':
                Sequenza<Cerchio> sc = new Sequenza<Cerchio>();
                double r;
                while ((r = in.readDouble("Inserire la lunghezza del raggio" +
                    " (0 per terminare) ")) != 0)
                    sc.add(new Cerchio(r));
                seq = sc;
                break;

            case 'q':
                Sequenza<Quadrato> sq = new Sequenza<Quadrato>();
                int l;
                while ((l = in.readInt("Inserire la lunghezza del lato" +
                    " (0 per terminare) ")) != 0)
                    sq.add(new Quadrato(l));
        }
    }
}

```

Nella
che la
ha scri
proprie
di cont
Pertan
è garan

Eser

6.28 I

fi
F

6.29 C

P
le

N
zi
fe
di
ne

```

    seq = sq;
    break;
}

//stampa delle aree
for (Figura f : seq)
    out.println(f.getArea());

}
}

```

Nella prima soluzione che abbiamo delineato, il programmatore utilizza il cast sulla base del fatto che la sequenza è stata costruita inserendo esclusivamente figure: questa proprietà è nota a chi ha scritto il programma ma non viene controllata dal compilatore. Nella seconda soluzione, la proprietà viene esplicitata nella dichiarazione della variabile `seq`; questo permette al compilatore di controllare che effettivamente nella sequenza vengano inserite figure e non altri tipi di oggetti. Pertanto, questa seconda soluzione offre un livello di sicurezza per quanto riguarda i tipi che non è garantito nella prima, prevenendo così un insieme significativo di errori di programmazione.

Esercizi

- 6.28 Implementate l'applicazione `TestSequenzeFigure`, in modo che visualizzi le aree delle figure inserite. Dichiarate la variabile `seq` di tipo `Sequenza<?>` e utilizzate un cast al tipo `Figura` all'interno del ciclo `for`.
- 6.29 Confrontate l'implementazione di `TestSequenzeFigure` che avete scritto per l'esercizio precedente, con quella fornita nel testo. Provate ad aggiungere, prima del ciclo `for` finale, le istruzioni

```

Sequenza<String> ss = new Sequenza<String>();
ss.add("pippo");
seq = ss;

```

Nell'implementazione fornita nel testo, questa modifica provocherà un errore in compilazione; nell'implementazione che utilizza il cast provocherà un errore in esecuzione. È preferibile riscontrare un problema presente nel codice nella fase di compilazione o in quella di esecuzione? Tenete presente che nella maggior parte dei casi i problemi relativi ai tipi non sono evidenti come in quest'esempio.

6.17 Tipi generici e vincoli sugli argomenti

È possibile costruire sequenze di oggetti di un qualunque tipo. In altre parole, non vi sono vincoli sul tipo argomento della classe `Sequenza`. Ciò è indicato nell'intestazione della classe e nella relativa documentazione da

```
class Sequenza<E>
```

dove `E` (o un qualunque altro identificatore racchiuso tra i simboli di minore e maggiore) indica la possibilità di fornire come argomento un qualunque tipo.

In molte situazioni, il tipo argomento deve soddisfare determinati requisiti. Consideriamo, ad esempio la classe `SequenzaOrdinata`, presente nel package `prog.utili`. Ricordiamo che le sue istanze sono sequenze ordinate di oggetti del tipo argomento. Un requisito indispensabile per poter costruire sequenze ordinate è che esista una relazione di ordine totale tra gli elementi del tipo argomento. Come abbiamo visto, questa proprietà viene definita implementando l'interfaccia `Comparable`. In altre parole, è possibile costruire sequenze ordinate di un tipo `E`, solo se `E` è un sottotipo di `Comparable<E>`. Ciò viene dichiarato nell'intestazione (e nella documentazione) della classe `SequenzaOrdinata` stabilendo un vincolo sul tipo parametro formale che deve essere sottotipo di `Comparable<E>` come

```
class SequenzaOrdinata<E extends Comparable<E>>
```

In questo contesto la parola riservata `extends` significa “è sottotipo”: il tipo argomento `E` può essere un qualunque tipo riferimento sottotipo di `Comparable<E>`. Dunque `E` deve essere una classe che implementa `Comparable<E>`.¹⁷

In realtà l'intestazione della classe `SequenzaOrdinata` è più complicata di quella indicata sopra. In particolare essa utilizza, tra l'altro, il segnaposto `?` con un vincolo

```
public class SequenzaOrdinata<E extends Comparable<? super E>>
```

In base a quanto indicato, il tipo argomento deve essere un qualunque sottotipo di `Comparable<? super E>`. Analizziamo il caso, più frequente, in cui come tipo argomento indichiamo il nome di una classe, supponiamo `A`; possiamo definire `SequenzaOrdinata<A>` a patto che `A` implementi l'interfaccia `Comparable`, dove `B` è un qualunque supertipo di `A` (compreso `A` stesso). Ad esempio, possiamo costruire `SequenzaOrdinata<Frazione>` in quanto la classe `Frazione` implementa l'interfaccia `Comparable<Frazione>` (in questo caso `B` coincide con `A`, cioè con `Frazione` stessa). Supponiamo di disporre di una sottoclassse `FrazioneSpeciale` di `Frazione`. Poiché `Frazione` implementa `Comparable<Frazione>`, grazie al meccanismo dell'ereditarietà, `FrazioneSpeciale` è sottotipo di `Comparable<Frazione>`. Possiamo dire direttamente che `FrazioneSpeciale` implementa `Comparable<Frazione>`: l'implementazione viene ereditata da `Frazione` o è propria della classe `FrazioneSpeciale`, a seconda che il metodo `boolean compareTo(Frazione altra)` non sia o sia ridefinito in `FrazioneSpeciale`.

¹⁷ E può anche essere un'interfaccia che estende `Comparable<E>`.

In og...
te di...
po Se...
caso

6.18

È im...
U...
In pa...
parar...
viam...
e Sec...
Ques...
verra...
una c...
parar...

L...
comp...
di inc...
al tip...
gener...
Sequ...

s...
o con...
(S...

sono...
di cor...
C...
gli ar...
ray: a...
Sequ...
quind...
l'asse...

Ret...

¹⁸ I...
vengon...

¹⁹ Q...

In ogni caso, la classe `FrazioneSpeciale` fornisce un metodo di ordinamento che permette di confrontare le proprie istanze con quelle di `Frazione`. Pertanto, possiamo definire il tipo `SequenzaOrdinata<FrazioneSpeciale>`. Rifacendoci ai nomi utilizzati sopra, in questo caso A è `FrazioneSpeciale`, mentre B è `Frazione`.

6.18 Tipi generici e compilazione

È importante, a questo punto, precisare una caratteristica fondamentale dei tipi generici in Java.

Un tipo generico come `Sequenza<E>`, definisce un'unica classe e non una famiglia di classi. In particolare, il compilatore genera un unico file corrispondente alla classe `Sequenza`: tutti i tipi parametrizzati ottenuti dal tipo generico corrispondono a quest'unica classe. Pertanto, se scriviamo un'applicazione che utilizza, ad esempio, `Sequenza<String>`, `Sequenza<Frazione>` e `Sequenza<Cerchio>`, non vi saranno tre copie della classe `Sequenza`, ma una sola classe.¹⁸ Questa caratteristica è cruciale quando si considerano i metodi statici (e i campi statici, che verranno analizzati nella parte relativa all'implementazione delle classi): un metodo statico di una classe generica è eseguito dall'intera classe e non da una sua particolare istanza o versione parametrizzata.

Le informazioni sul tipo argomento specificato nel tipo parametrizzato sono utilizzate dal compilatore per controllare l'uso corretto dei tipi. Come abbiamo già sottolineato, ciò permette di individuare già in fase di compilazione errori relativi all'uso dei tipi. Le informazioni relative al tipo argomento vengono rimosse dal compilatore¹⁹ e, dunque, non sono presenti nel bytecode generato. Pertanto, nella fase di esecuzione, un oggetto costruito invocando il tipo parametrizzato `Sequenza<String>` è, semplicemente, di tipo `Sequenza`. Quindi, espressioni come:

```
s instanceof Sequenza<String>
```

o come

```
(Sequenza<String>) s
```

sono del tutto prive di significato e generano messaggi di avvertimento o di errore durante la fase di compilazione.

Concludiamo il paragrafo segnalando che vi sono profonde differenze tra i tipi generici e gli array. Le relazioni gerarchiche tra i tipi riferimento si riflettono sui corrispondenti tipi array: ad esempio `Rettangolo[]` è supertipo di `Quadrato[]` mentre, come abbiamo osservato, `Sequenza<Rettangolo>` non è supertipo di `Sequenza<Quadrato>`. Si noti che `Object[]` è quindi un supertipo di tutti gli array. Avendo definito `aq` di tipo `Quadrato[]`, è possibile scrivere l'assegnamento

```
Rettangolo[] ar = aq;
```

¹⁸ I *template* del linguaggio C++, apparentemente analoghi ai tipi generici di Java, sono "schemi" di classi, dai quali vengono generate differenti classi sostituendo il tipo parametro con il tipo argomento.

¹⁹ Quest'operazione viene detta *erasure*, in italiano *cancellazione*.

In questo modo, si potrebbero scrivere istruzioni che tramite il riferimento `ar` tentino di portare l'array riferito da `aq` in uno stato inconsistente, provocando un errore in esecuzione. Nel bytecode generato dal compilatore, per gli array vengono riportate le informazioni relative al tipo base. Quando viene costruito un oggetto di tipo array, esso contiene le informazioni relative al proprio tipo base. Ad esempio, invocando il costruttore `new Quadrato[5]` viene predisposto un contenitore con 5 posizioni di tipo riferimento a `Quadrato`.

Esercizi

- 6.30 Scrivete un'applicazione che legga i lati di 5 quadrati, memorizzi gli oggetti corrispondenti in un array di tipo `Quadrato`, e infine scandisca l'array invocando il metodo `toString` di ogni oggetto in esso contenuto.
- 6.31 Supponendo che il riferimento all'array costruito dall'applicazione dell'esercizio precedente si chiami `aq`, aggiungete, alla fine del metodo `main`, le seguenti istruzioni:

```
Rettangolo[] ar = aq;
ar[3] = new Quadrato(4);
ar[4] = new Rettangolo(3, 1);
```

Come vi aspettate che si comporti l'ambiente Java con questo codice? È possibile compilare? È possibile eseguire? Se invece di usare un array di quadrati aveste usato una `Sequenza<Quadrato>` il comportamento sarebbe stato lo stesso?

- 6.32 Considerate l'applicazione `OccorrenzeParole` del Paragrafo 6.11. Tale applicazione utilizza istanze della classe `Occorrenza` per contare le occorrenze di una stringa e la classe generica `Sequenza` per memorizzare la tavola delle occorrenze. Utilizzando la classe `OccorrenzaOrdinata` e la classe `SequenzaOrdinata` del package `prog.utili` riscrivete l'applicazione `OccorrenzeParole` in maniera che produca l'elenco delle parole in ordine alfabetico. Si osservi che la classe `OccorrenzaOrdinata` prevede un tipo argomento che rispetti il vincolo `E extends Comparable<? super E>`, in altre parole prevede come parametro `E` il nome di una classe che implementi l'interfaccia `Comparable<T>` dove `T` è una superclasse di `E` (eventualmente `E` stessa). Inoltre, implementa a sua volta l'interfaccia `Comparable<OccorrenzaOrdinata<E>>` (e quindi può essere utilizzata come tipo argomento di `SequenzaOrdinata`). Infine, tale classe estende `Occorrenza<E>` e fornisce, oltre ai metodi di tale classe, solo i metodi

```
public int compareTo(OccorrenzaOrdinata<E> altra)
public boolean equals(OccorrenzaOrdinata<E> altra)
```

fornisce, infine, due costruttori analoghi a quelli di `Occorrenza`.

- 6.33 Ispirandovi all'Esercizio 6.32 riscrivete l'applicazione `OccorrenzeLettere` sviluppata per l'Esercizio 6.19 in modo che produca l'elenco in ordine alfabetico.

Parte II

Implementazione degli oggetti

Implementazione delle classi

Nella prima parte del testo abbiamo utilizzato classi già definite, come ad esempio la classe `String` o la classe `ConsoleInputManager`, e istanze di tali classi. Abbiamo anche definito qualche classe, ma solo per poter scrivere applicazioni eseguibili, basate su semplici metodi `main`. Inizieremo ora a studiare come implementare una classe, in modo che possa essere poi utilizzata da altre classi.

7.1 Classi e oggetti

Possiamo immaginare una classe come una *ricetta* che descrive come sono fatti determinati oggetti o come una *fabbrica* che permette di costruire oggetti con precise caratteristiche. Ogni oggetto è caratterizzato da un insieme di *dati*, che ne descrivono lo *stato*, e da un insieme di *metodi*, che ne descrivono il *comportamento*, cioè ciò che l'oggetto può fare. Ad esempio lo stato di un oggetto della classe `String` è la stringa di caratteri rappresentata, mentre i comportamenti sono i metodi che abbiamo presentato, come `length`, `toUpperCase`, etc. Quando vogliamo scrivere una classe dobbiamo pensare prima di tutto a ciò che gli oggetti della classe devono rappresentare e, di conseguenza, anche a ciò che questi oggetti devono essere in grado di fare o, in altre parole, ai metodi di cui devono disporre. La classe dev'essere progettata, per quanto possibile, indipendentemente dalle applicazioni che la utilizzeranno: ad esempio chi ha progettato la classe `String` non poteva sapere che l'avremmo usata per scrivere l'applicazione `Cornice`. Una classe ben progettata potrà essere utilizzata successivamente per svariate applicazioni.

Una volta che abbiamo deciso *che cosa* deve fare la classe (cioè che cosa rappresenta e quali metodi fornisce) possiamo decidere *come* realizzarla. Chi la utilizzerà dovrà sapere *che cosa* essa è in grado di fare, ma non sarà tenuto a conoscere *come* è stata realizzata. In pratica, per chi la utilizza, la classe dev'essere come una scatola, di cui è visibile solo la parte esterna (prototipi dei metodi e costruttori). Una stessa classe, come vedremo, potrebbe essere implementata in diversi modi.

Il primo esempio di questo capitolo è la classe `Frazione`. Abbiamo già fornito la *specifica del suo comportamento*, ossia *che cosa fa* e *che cosa fanno* i suoi oggetti, descrivendone costrut-

tori e metodi nel Paragrafo 3.1. Studiamo ora *come* implementare la classe, cioè come realizzare il comportamento indicato e codificarlo in linguaggio Java.

L'implementazione della classe viene scritta all'interno del *corpo* della classe, cioè all'interno della parte di codice racchiusa tra la parentesi graffa aperta dopo l'intestazione della classe e la parentesi graffa chiusa finale.

Sulla base della specifica cominciamo a scrivere la struttura della classe:

```
public class Frazione {
    ...
    public Frazione(int x, int y) {
        ...
    }
    public Frazione(int x) {
        ...
    }
    public Frazione piu(Frazione f) {
        ...
    }
    public Frazione meno(Frazione f) {
        ...
    }
    public Frazione per(Frazione f) {
        ...
    }
    public Frazione diviso(Frazione f) {
        ...
    }
    public boolean equals(Frazione f) {
        ...
    }
    public boolean isMinore(Frazione f) {
        ...
    }
    public boolean isMaggiore(Frazione f) {
```

}

pub

pub

}

pub

}

pub

}

All'interno verranno definite le classi e le strutture dati necessarie per rappresentare numeri razionali.

La classe Frazione ha un attributo privato.

Scriviamo una nuova classe per la seconda operazione di creazione di frazioni.

(1) fare in modo che la classe Frazione possa essere creata solo con due numeri interi diversi da zero.

(2) scrivere i metodi per confrontare frazioni.

Approfondire i casi di leggibilità.

```

    ...
}

public int getNumeratore() {
    ...
}

public int getDenominatore() {
    ...
}

public String toString() {
    ...
}
}

```

All'interno del corpo, oltre ai costruttori e ai metodi di cui abbiamo evidenziato le intestazioni, verranno scritti i *campi*, cioè i dati che definiscono lo *stato* del singolo oggetto della classe.

Una frazione è definita da due numeri interi, il numeratore e il denominatore. Pertanto nella classe **Frazione** definiamo due campi di tipo **int**, denominati **num** e **den**. Poiché la rappresentazione dell'oggetto riguarda solo chi progetta la classe, ma non chi la utilizza, è opportuno nascondere questi campi al codice esterno alla classe. A tale scopo dichiariamo che i due campi sono *privati* scrivendo la parola riservata **private** all'inizio della dichiarazione.

I campi sono anche chiamati *variabili di istanza*, in quanto sono variabili associate alla singola istanza della classe: ogni istanza della classe **Frazione** avrà due campi propri, chiamati **num** e **den**.

La dichiarazione che possiamo inserire all'inizio del corpo della classe¹ è dunque:

```
private int num, den;
```

Scriviamo ora il costruttore con due argomenti: deve ricevere due numeri interi e deve costruire una nuova frazione che abbia come numeratore il primo argomento e come denominatore il secondo. Possiamo pensare che una classe, quando se ne richiama il costruttore in un'espressione di creazione, lavori in questo modo:

- (1) fabbrica prima di tutto un oggetto contenente i campi definiti nella classe, inizializzati con valori fissati (nel caso del tipo **int** sono inizializzati a zero);
- (2) successivamente esegue le istruzioni scritte nel codice del costruttore con le quali, ad esempio, è possibile assegnare valori differenti ai campi.

Approfondiremo questi punti più avanti.

¹ I campi possono essere scritti in qualsiasi punto all'interno del corpo della classe. Tuttavia, per rendere il codice più leggibile, li scriveremo sempre all'inizio del corpo della classe, seguiti dai costruttori e, infine, dai metodi.

Nel nostro caso, alla chiamata del costruttore viene creato un nuovo oggetto; dovremo assegnare ai suoi campi i valori dei due argomenti del costruttore. Pertanto il corpo del costruttore è costituito semplicemente da due assegnamenti:

```
public Frazione(int x, int y) {
    num = x;
    den = y;
}
```

Il secondo costruttore può essere scritto in maniera analoga.

A questo punto siamo in grado di capire meglio che cosa accade quando la Java Virtual Machine crea un oggetto. Consideriamo ad esempio le seguenti istruzioni:

```
Frazione f1 = new Frazione(1,2);
Frazione f2 = new Frazione(2,3);
```

Come evidenziato nella Figura 7.1, la Java Virtual Machine ha riservato lo spazio per memorizzare due oggetti di tipo `Frazione`. Ognuno degli oggetti contiene, oltre a informazioni di controllo indispensabili alla Java Virtual Machine, lo spazio necessario a memorizzare lo stato che, nel nostro caso, è costituito dai valori dei due campi di tipo `int`. Si osservi come ogni oggetto possieda propri campi `num` e `den`. Osserviamo anche che l'oggetto contiene solo informazioni relative al-

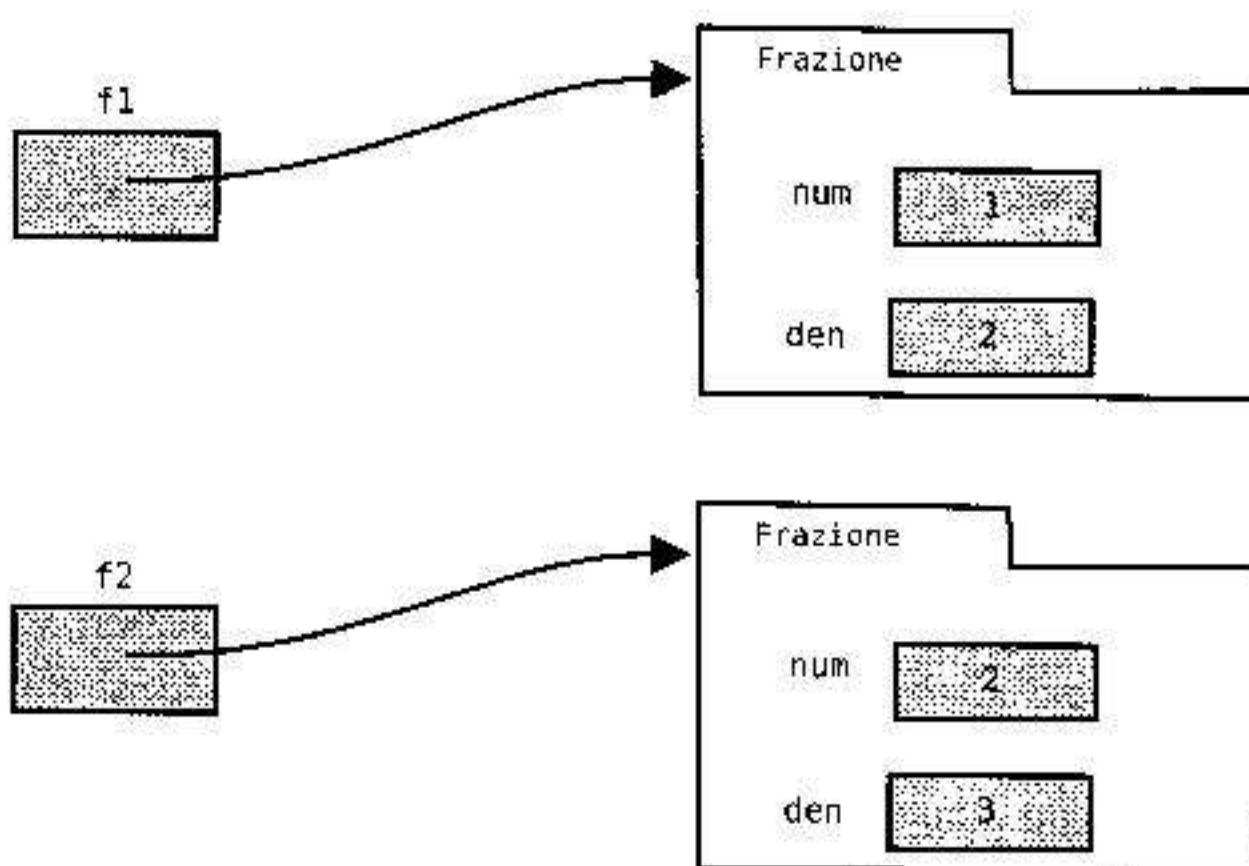


Figura 7.1 I due oggetti di tipo `Frazione`.

lo stato; i metodi, in particolare il loro bytecode, sono memorizzati nella classe (nel file `.class` per la precisione). Quando un oggetto deve eseguire un metodo in risposta a un messaggio, la Java Virtual Machine recupera il codice da eseguire nel file `.class` e lo fornisce all'oggetto, che quindi lo esegue sulla base del proprio stato. Ad esempio, le stringhe restituite dalle espressioni `f1.toString()` e `f2.toString()` sono diverse in quanto lo stato dei due oggetti è differente, ma il codice eseguito dagli oggetti riferiti da `f1` e `f2` è il medesimo.

Codifichiamo ora i metodi della classe. Iniziamo a esaminare il metodo `per`. Ricordiamo che questo metodo è eseguito da un oggetto, cioè da una frazione. Secondo la specifica, il metodo deve ricevere come argomento un'altra frazione, e restituire come risultato il riferimento a una nuova frazione data dal prodotto tra il valore dell'oggetto che esegue il metodo e l'argomento. Pertanto, possiamo immaginare di svolgere i seguenti passi:

- calcola il numeratore della nuova frazione
- calcola il denominatore della nuova frazione
- costruisci la nuova frazione
- restituisci un riferimento a essa

Il numeratore della nuova frazione è dato dal prodotto dei numeratori delle due frazioni. Per ottenere i numeratori è sufficiente accedere ai campi `num` delle due frazioni. Tali campi sono accessibili perché il metodo si trova all'interno della classe `Frazione` (al contrario, essendo dichiarati `private`, essi non sono accessibili a tutto il codice scritto esternamente alla classe). Per accedere a un campo si utilizza la notazione

`riferimento_a_oggetto.nome_campo`

Ad esempio, se `f` è il nome di un riferimento all'altra frazione, per accedere al campo `num` è sufficiente scrivere:

`f.num`

Quando all'interno di un metodo vogliamo riferirci all'oggetto stesso che esegue il metodo, possiamo utilizzare il riferimento `this`. In pratica la scrittura:

`this.num`

indica il campo `num` di *questo* oggetto, cioè dell'oggetto che esegue il metodo.

Il numeratore e il denominatore della nuova frazione possono essere memorizzati in due variabili `n` e `d` dichiarate localmente al metodo `per`:

```
n = this.num * f.num;
d = this.den * f.den;
```

A questo punto costruiamo la nuova frazione richiamando il costruttore con due argomenti e memorizziamo il risultato, cioè il riferimento al nuovo oggetto, in una variabile locale `g` di tipo `Frazione`:

```
g = new Frazione(n, d);
```

Infine il metodo `per` deve restituire il risultato, ossia il riferimento alla nuova frazione, appena memorizzato nella variabile `g`. Per restituire un risultato si utilizza l'istruzione `return` seguita dal risultato da restituire. In questo caso possiamo scrivere:

```
return g;
```

Il metodo è quindi:

```
public Frazione per(Frazione f) {
    int n = this.num * f.num;
    int d = this.den * f.den;
    Frazione g = new Frazione(n, d);
    return g;
}
```

Le variabili `n`, `d` e `g` sono definite localmente al metodo e, per questo motivo, sono anche dette *variabili locali*. Esse sono utilizzate per la memorizzazione temporanea di valori durante l'esecuzione del metodo. In altre parole, forniscono un supporto all'attività del metodo. Quando l'esecuzione del metodo termina (istruzione `return`), non hanno più alcuna ragione di esistere e, di conseguenza, vengono distrutte. Approfondiremo questi aspetti nella parte riguardante il comportamento della Java Virtual Machine e la gestione della memoria durante l'esecuzione.

Si può anche evitare di introdurre la variabile `g` scrivendo, dopo la parola riservata `return`, l'espressione che fornisce il risultato da restituire:

```
return new Frazione(n, d);
```

Il testo del metodo diventa così:

```
public Frazione per(Frazione f) {
    int n = this.num * f.num;
    int d = this.den * f.den;
    return new Frazione(n, d);
}
```

Volendo, si potrebbe scrivere il metodo in maniera più compatta, senza introdurre le variabili `n` e `d`. Tuttavia è preferibile evitare una scrittura troppo compatta del codice, perché ne ridurrebbe notevolmente la leggibilità.

Gli altri metodi per il calcolo di operazioni aritmetiche possono essere sviluppati in maniera analoga.

Scriviamo ora il metodo `equals`. Esso deve ricevere come parametro una frazione e restituire la costante `true` nel caso il valore del parametro sia uguale a quello dell'oggetto che esegue il metodo, `false` in caso contrario.

Per sviluppare questo metodo dobbiamo chiederci che cosa intendiamo per uguaglianza di frazioni. Infatti frazioni differenti possono rappresentare lo stesso valore numerico: ad esempio $\frac{3}{4}$, $\frac{6}{8}$ e $-\frac{3}{-4}$ sono lo stesso numero. Dunque l'uguaglianza tra due frazioni non può essere decisa semplicemente sulla base dell'uguaglianza dei rispettivi campi `num` e `den`. Esistono diverse strategie per risolvere questo problema.

Una di queste consiste nel sottrarre una frazione dall'altra mediante il metodo `meno`, e controllare poi che il numeratore della frazione risultante sia nullo:

```

calcola la differenza tra le due frazioni
SE il numeratore della frazione risultante è uguale a zero
    ALLORA
        restituisci il valore true
    ALTRIMENTI
        restituisci il valore false
FINESE

```

Supponendo di aver memorizzato in una variabile `f` il riferimento alla frazione da sottrarre, per calcolare la differenza tra le due frazioni è sufficiente richiamare il metodo `meno` dell'oggetto in esecuzione fornendo come argomento `f`. Il risultato è un riferimento a una nuova frazione, che memorizziamo in una variabile locale `g`. Per richiamare un metodo dell'oggetto in esecuzione possiamo servirci del riferimento `this`:

```
Frazione g = this.meno(f);
```

A questo punto dobbiamo effettuare una selezione in base al valore del numeratore della frazione riferita da `g`, cioè in base al campo `g.num`, utilizzando l'operatore `==` per la verifica di uguaglianza. In particolare la selezione avverrà in base alla condizione `g.num == 0`. Utilizzando il costrutto `if`, possiamo facilmente scrivere il metodo come segue:

```

public boolean equals(Frazione f) {
    Frazione g = this.meno(f);
    if (g.num == 0)
        return true;
    else
        return false;
}

```

I metodi `isMinore` e `isMaggiore` possono essere codificati in maniera analoga. Nel metodo `isMinore` dobbiamo controllare che la differenza tra le due frazioni sia negativa. Ci sono diversi modi per effettuare questo controllo; uno dei più semplici consiste nel verificare che i segni del numeratore e del denominatore della differenza siano opposti:

```

public boolean isMinore(Frazione f) {
    Frazione g = this.meno(f);
    if ((g.num < 0 && g.den > 0) || (g.num > 0 && g.den < 0))
        return true;
    else
        return false;
}

```

Il metodo `toString` deve restituire una stringa che rappresenti la frazione. La stringa può essere ottenuta semplicemente concatenando una stringa che rappresenti il numeratore con la stringa

"/", e con una stringa che rappresenti il denominatore. Utilizzando l'operatore + tra stringhe possiamo scrivere:

```
String risultato = this.num + "/" + this.den;
return risultato;
```

In realtà il riferimento `this` può essere omesso, sottintendendo così che ci riferiamo ai campi dell'oggetto stesso. Inoltre possiamo evitare di introdurre la variabile `risultato`, cioè possiamo scrivere il metodo come:

```
public String toString() {
    return num + "/" + den;
}
```

Questo metodo `toString` non è del tutto corretto: se il denominatore contiene zero, la frazione non ha senso. In questo caso è opportuno che venga fornita, come `risultato`, la stringa "impossibile".

```
public String toString() {
    if (den == 0)
        return "impossibile";
    else
        return num + "/" + den;
}
```

Dato che i campi `num` e `den` sono privati, dall'esterno della classe non è possibile accedere direttamente al valore del numeratore e del denominatore. È quindi utile prevedere altri due metodi che forniscono tali valori: `getNumeratore` e `getDenominatore`. Come esempio riportiamo il metodo `getNumeratore`:

```
public int getNumeratore() {
    return num;
}
```

Ecco il testo completo della classe così costruita:

```
public class Frazione {
    //CAMPI
    private int num, den;

    //COSTRUTTORI
    public Frazione(int x, int y) {
        num = x;
        den = y;
    }
```

```
public Frazione(int x) {
    num = x;
    den = 1;
}

//METODI
public Frazione piu(Frazione f) {
    int n = this.num * f.den + this.den * f.num;
    int d = this.den * f.den;
    return new Frazione(n, d);
}

public Frazione meno(Frazione f) {
    int n = this.num * f.den - this.den * f.num;
    int d = this.den * f.den;
    return new Frazione(n, d);
}

public Frazione per(Frazione f) {
    int n = this.num * f.num;
    int d = this.den * f.den;
    return new Frazione(n, d);
}

public Frazione diviso(Frazione f) {
    int n = this.num * f.den;
    int d = this.den * f.num;
    return new Frazione(n, d);
}

public boolean equals(Frazione f) {
    Frazione g = this.meno(f);
    if (g.num == 0)
        return true;
    else
        return false;
}

public boolean isMinore(Frazione f) {
    Frazione g = this.meno(f);
```

```

    if ((g.num < 0 && g.den > 0) || (g.num > 0 && g.den < 0))
        return true;
    else
        return false;
}

public boolean isMaggiore(Frazione f) {
    Frazione g = this.meno(f);
    if ((g.num < 0 && g.den < 0) || (g.num > 0 && g.den > 0))
        return true;
    else
        return false;
}

public String toString() {
    if (den == 0)
        return "impossibile";
    else
        return num + "/" + den;
}

public int getNumeratore() {
    return num;
}

public int getDenominatore() {
    return den;
}
}

```

Ecco, infine, una classe di prova per i metodi della classe **Frazione**:

```

import prog.io.*;

public class ProvaFrazione {

    public static void main(String args[]) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();
        int n, d;

        n = in.readInt("Prima Frazione: numeratore? ");

```

```
d = in.readInt("Prima Frazione: denominatore? ");
Frazione f1 = new Frazione(n, d);

n = in.readInt("Seconda Frazione: numeratore? ");
d = in.readInt("Seconda Frazione: denominatore? ");
Frazione f2 = new Frazione(n, d);

out.println("La prima frazione è uguale a " + f1.toString());
out.println("La seconda frazione è uguale a " + f2.toString());

if (f1.equals(f2))
    out.println("Le due frazioni sono uguali");
else
    out.println("Le due frazioni sono diverse");
if (f1.isMaggiore(f2))
    out.println(f1.toString() + " è maggiore di " + f2.toString());
else
    out.println(f1.toString() + " non è maggiore di " + f2.toString());
if (f1.isMinore(f2))
    out.println(f1.toString() + " è minore di " + f2.toString());
else
    out.println(f1.toString() + " non è minore di " + f2.toString());

Frazione f3;

f3 = f1.piu(f2);
out.print(f1.toString() + " + " + f2.toString() + " = ");
out.println(f3.toString());

f3 = f1.per(f2);
out.print(f1.toString() + " * " + f2.toString() + " = ");
out.println(f3.toString());

f3 = f1.meno(f2);
out.print(f1.toString() + " - " + f2.toString() + " = ");
out.println(f3.toString());

f3 = f1.diviso(f2);
out.print(f1.toString() + " : " + f2.toString() + " = ");
out.println(f3.toString());
}
```

7.2 La classe Frazione: alcuni miglioramenti

Rivediamo alcuni aspetti della classe `Frazione` implementata nel paragrafo precedente.

Costruttori

Nella classe sono stati definiti i seguenti costruttori:

```
public Frazione(int x, int y) {
    num = x;
    den = y;
}

public Frazione(int x) {
    num = x;
    den = 1;
}
```

Possiamo osservare che il secondo costruttore può essere simulato dal primo fornendo il valore 1 in luogo dell'argomento corrispondente al denominatore. In sostanza, il secondo costruttore potrebbe delegare il proprio compito al primo fornendogli come argomenti x e 1.

Quando in una classe c'è più di un costruttore, è possibile che un costruttore ne chiami un altro utilizzando il riferimento `this`, seguito dalla lista di argomenti da fornire al costruttore chiamato. La chiamata del costruttore *dev'essere la prima istruzione* del codice del costruttore chiamante. In sostanza, un costruttore può chiedere a un altro costruttore della stessa classe di fabbricare come prima cosa il nuovo oggetto (chiamata con il riferimento `this` nella prima istruzione), e può poi adattare l'oggetto alle proprie esigenze (nelle istruzioni successive).

Nel nostro caso, il costruttore con un parametro può chiedere a quello con due di costruire l'oggetto. Una volta che l'oggetto è stato fabbricato, non c'è bisogno di alcuna modifica. Pertanto possiamo riscrivere il costruttore con un parametro come:

```
public Frazione(int x) {
    this(x, 1);
}
```

Il metodo diviso

Abbiamo visto che è possibile costruire oggetti che dal punto di vista matematico non rappresentano frazioni. Ad esempio, se richiamiamo il costruttore con due parametri fornendo come argomenti 8 e 0 o 1 e 0, in ambedue i casi ciò che viene rappresentato non è una frazione. Nel metodo `toString` abbiamo tenuto conto di questo, facendo in modo che in tali casi venga restituita la stringa "impossibile".

I metodi `piu`, `meno` e `per`, nel caso l'oggetto in esecuzione o l'argomento fornito non rappresenti una frazione, producono un oggetto che a sua volta non rappresenta una frazione. Ciò

può essere facilmente verificato osservando che, quando uno dei due denominatori è zero, anche il denominatore della frazione risultante è zero.

Esaminiamo invece il metodo `diviso`:

```
public Frazione diviso(Frazione f) {
    int n = this.num * f.den;
    int d = this.den * f.num;
    return new Frazione(n, d);
}
```

Se l'oggetto che esegue il metodo non rappresenta una frazione, cioè se il suo campo `den` contiene zero, anche il campo `den` dell'oggetto risultante conterrà zero. Se invece l'oggetto che esegue il metodo rappresenta una frazione, ad esempio $1/2$, e l'oggetto fornito tramite il parametro non rappresenta una frazione, ad esempio contiene 1 nel campo `num` e 0 nel campo `den`, il risultato prodotto dal metodo è la frazione $0/2$, di valore nullo. In pratica il risultato della divisione di $1/2$ per "impossibile" sarebbe 0 anziché "impossibile".

Per risolvere questo problema modifichiamo il metodo introducendo una selezione: se il campo `den` dell'oggetto fornito tramite il parametro contiene zero, costruiamo un nuovo oggetto che non rappresenta alcuna frazione, altrimenti procediamo effettuando i calcoli:

```
public Frazione diviso(Frazione f) {
    if (f.den == 0)
        return new Frazione(0, 0);
    else {
        int n = this.num * f.den;
        int d = this.den * f.num;
        return new Frazione(n, d);
    }
}
```

Per esercizio, riesamineate i tre metodi forniti nella classe `Frazione` per il confronto di frazioni. Controllate se il loro comportamento, qualora almeno uno dei due oggetti da esaminare non rappresenti una frazione, è sempre corretto. Eventualmente riscriveteli in modo corretto.

Il metodo `toString`

Nel caso il campo denominatore contenga 1, il metodo `toString` restituisce una stringa che riporta il risultato sotto forma di frazione, come ad esempio "4/1". In questo caso sarebbe opportuno restituire direttamente una stringa come "4", senza l'indicazione del denominatore. Il problema può essere risolto introducendo un'ulteriore selezione:

```
SE il denominatore è uguale a 0
ALLORA
    restituisci "impossibile"
ALTRIMENTI
```

```

SE il denominatore è uguale a 1
ALLORA
    restituisci il valore del numeratore
ALTRIMENTI
    restituisci la stringa della forma numeratore/denominatore
FINESE
FINESE

```

Un'immediata codifica di questo schema potrebbe essere:

```

public String toString() {
    if (den == 0)
        return "impossibile";
    else if (den == 1)
        return num;
    else
        return num + "/" + den;
}

```

Questo codice non è corretto. Infatti l'istruzione `return`, con la quale un metodo restituisce a chi l'ha chiamato un valore, dev'essere seguita dal risultato (anche sotto forma di espressione) da restituire. Poiché nell'intestazione abbiamo dichiarato che il metodo restituisce un risultato di tipo `String`, ogni istruzione `return` che compare nel metodo dovrà restituire un valore `String`. In questo caso la seconda istruzione `return` restituisce invece il valore di `num`, cioè un `int`.

Per risolvere questo problema dobbiamo trasformare il valore intero, memorizzato in `num`, in una stringa di caratteri. A tal fine possiamo utilizzare il metodo statico `valueOf` della classe `String`:

```

public String toString() {
    if (den == 0)
        return "impossibile";
    else if (den == 1)
        return String.valueOf(num);
    else
        return num + "/" + den;
}

```

Esercizi

- 7.1 Aggiungete alla classe `Frazione` un nuovo metodo di nome `inversa` che restituisca il riferimento a una frazione inversa rispetto alla frazione memorizzata nell'oggetto. Se ad esempio l'oggetto contiene la frazione $5/3$, la frazione restituita dovrà essere $3/5$.

7.2 C
da
in

7.3 S
(s
st

7.3

Notiam
zioni. A
fornisse
tore e d
frazioni
la secon
confron

Prin

publ
si
nu
de
}

Per il ca
nome m
utile all
servizio
legato a
tale sco

priv

Esamin
codifica

- la
- la
- so
- u

Trattiam
essere p

- 7.2 Come si comporta il metodo sviluppato al punto precedente se la frazione rappresentata dall'oggetto non ha significato, cioè se il denominatore è nullo? Modificalo in modo che, in caso di denominatore nullo, la frazione risultante indichi ancora il valore impossibile.
- 7.3 Scrivete un'implementazione della classe `Rettangolo` descritta nei capitoli precedenti (senza fare riferimento alla gerarchia delle classi, ossia sviluppandola come classe a sé stante, a prescindere dalla classe `Figura`).

7.3 Una nuova implementazione della classe Frazione

Notiamo che, per come è stata definita la classe `Frazione`, non vengono mai operate semplificazioni. Ad esempio sarebbe opportuno che in luogo della stringa "32/128" il metodo `toString` fornisse "1/4". Ricordiamo che, per semplificare una frazione, è necessario dividere numeratore e denominatore per il loro massimo comun divisore. Possiamo decidere di semplificare le frazioni prima di visualizzarle, oppure di semplificarle al momento della costruzione. Scegliamo la seconda strategia; questo ci permetterà anche di scrivere alcuni metodi (ad esempio quelli di confronto) in maniera più semplice.

Prima di tutto riscriviamo il costruttore con due argomenti secondo il seguente schema:

```
public Frazione(int x, int y) {
    sia m il massimo comun divisore di x e y;
    num = x / m;
    den = y / m;
}
```

Per il calcolo del massimo comun divisore introduciamo, nella classe `Frazione`, un metodo di nome `mcd`. Questo metodo non svolge azioni legate a un singolo oggetto, ma esegue un calcolo utile alla classe stessa durante la costruzione di nuovi oggetti. In sostanza, questo metodo è un servizio che la classe offre a se stessa: è pertanto opportuno che sia dichiarato statico (perché non legato allo specifico oggetto) e privato (in quanto svolge esclusivamente un servizio interno). A tale scopo, nell'intestazione del metodo scriveremo i *modificatori private e static*:

```
private static int mcd(int a, int b)
```

Esamineremo più avanti come scrivere il codice del metodo `mcd`; completiamo invece la nuova codifica del costruttore. A tal fine dobbiamo tenere conto di altri due problemi:

- la possibilità di ricevere parametri negativi;
- la possibilità di ricevere il secondo parametro uguale a zero (ad esempio per un inserimento scorretto o come risultato dell'uso del metodo `diviso` in cui il parametro sia una frazione uguale a zero).

Trattiamo il primo problema decidendo che l'eventuale segno negativo della frazione dovrà essere posto, dopo la semplificazione, davanti al numeratore. Pertanto nel costruttore:

- all'inizio memorizziamo in una variabile boolean denominata `negativo` l'eventuale segno meno del risultato:

```
boolean negativo = (x < 0 && y > 0) || (x > 0 && y < 0);
```

- sostituiamo i valori dei parametri `x` e `y` con i rispettivi valori assoluti:

```
if (x < 0)
    x = - x;
if (y < 0)
    y = - y;
```

- calcoliamo il massimo comun divisore richiamando il metodo `mcd` (poiché non c'è ambiguità, possiamo richiamarlo indicandone semplicemente il nome, senza premettere il riferimento `Frazione` alla classe):

```
int m = mcd(x, y);
```

- dividiamo i parametri per il massimo comun divisore ripristinando l'eventuale segno meno davanti al numeratore:

```
if (negativo)
    num = - x / m;
else
    num = x / m;
den = y / m;
```

Il secondo parametro uguale a zero indica una situazione anomala. In questo caso decidiamo di porre direttamente a zero ambedue i campi della frazione, senza quindi svolgere la parte relativa alla semplificazione. Ecco il costruttore così modificato:

```
public Frazione(int x, int y) {
    if (y == 0) {
        num = 0;
        den = 0;
    } else {
        //memorizza il segno
        boolean negativo = (x < 0 && y > 0) || (x > 0 && y < 0);
        if (x < 0)
            x = - x; //elimina l'eventuale segno meno davanti a x
        if (y < 0)
            y = - y; //elimina l'eventuale segno meno davanti a y
    }
}
```

```

int m = mcd(x, y);
if (negativo)
    num = - x / m; //il segno viene memorizzato al numeratore
else
    num = x / m;
den = y / m;
}
}

```

In questa nuova implementazione, frazioni con lo stesso valore vengono memorizzate nello stesso modo. Ad esempio le frazioni $2/-4$, $-3/6$ e $-4/8$ saranno tutte rappresentate con -1 nel campo num e 2 nel campo den. Pertanto il metodo `equals` può essere realizzato verificando che, nelle due frazioni, sia i campi num sia i campi den contengano lo stesso valore. Ricordiamo che una frazione è l'oggetto stesso che esegue il metodo (riferimento `this`), mentre dell'altra è fornito il riferimento tramite il parametro (supponiamo di nome `f`). In sostanza dobbiamo verificare che valgano *ambedue* le condizioni `this.num == f.num` e `this.den == f.den`.

Pertanto, il metodo `equals` può essere riscritto come:

```

public boolean equals(Frazione f) {
    if (this.num == f.num && this.den == f.den)
        return true;
    else
        return false;
}

```

o, più semplicemente, come:

```

public boolean equals(Frazione f) {
    return this.num == f.num && this.den == f.den;
}

```

Si noti che il metodo gestisce correttamente anche il caso in cui almeno uno dei due oggetti non rappresenti una frazione.

Anche i metodi `isMinore` e `isMaggiore` possono essere semplificati. Nella nuova rappresentazione il denominatore ha sempre segno positivo; pertanto il segno di una frazione è uguale a quello del numeratore. Dunque i metodi possono essere riscritti come:

```

public boolean isMinore(Frazione f) {
    Frazione g = this.meno(f);
    return g.num < 0;
}

public boolean isMaggiore(Frazione f) {
    Frazione g = this.meno(f);
    return g.num > 0;
}

```

Osserviamo, infine, che qualora l'oggetto non rappresenti una frazione, sia il numeratore sia il denominatore contengono zero. Questo consente di evitare l'inserimento di controlli nel metodo `diviso`, come avevamo fatto in precedenza.

Sviluppiamo ora il metodo `mcd` per il calcolo del massimo comun divisore. A tale scopo ci basiamo sull'algoritmo di Euclide e, in particolare, sulla versione presentata nel Paragrafo 1.11:

variabili `x, y, resto`: numeri interi

```
leggi x, y
ESEGUI
    resto ← x MOD y
    x ← y
    y ← resto
QUANDO resto != 0
    scrivi x
```

```
}
```

```
publ
    th
}
```

Chiamando i parametri `a` e `b`, possiamo scrivere il metodo in questo modo:

```
private static int mcd(int a, int b) {
    int resto;
    do {
        resto = a % b;
        a = b;
        b = resto;
    } while (resto != 0);
    return a;
}
```

```
//ME
publ
    in
    in
    re
}
```

```
publ
    in
    in
    re
}
```

Ecco il testo completo della nuova versione della classe `Frazione`:

```
public class Frazione {
    //CAMPI
    private int num, den;

    //COSTRUTTORI
    public Frazione(int x, int y) {
        if (y == 0) {
            num = 0;
            den = 0;
        } else {
            //memorizza il segno
            boolean negativo = (x < 0 && y > 0) || (x > 0 && y < 0);
            if (x < 0)
                x = -x; //elimina l'eventuale segno meno davanti a x
            if (y < 0)
```

```
publ
    int
    int
    ret
}
```

```
publ
    int
    int
    ret
}
```

```
publ
    ret
}
```

```
y = - y; //elimina l'eventuale segno meno davanti a y

int m = mcd(x, y);
if (negativo)
    num = - x / m; //il segno viene memorizzato al numeratore
else
    num = x / m;
den = y / m;
}

public Frazione(int x) {
    this(x, 1);
}

//METODI
public Frazione piu(Frazione f) {
    int n = this.num * f.den + this.den * f.num;
    int d = this.den * f.den;
    return new Frazione(n, d);
}

public Frazione meno(Frazione f) {
    int n = this.num * f.den - this.den * f.num;
    int d = this.den * f.den;
    return new Frazione(n, d);
}

public Frazione per(Frazione f) {
    int n = this.num * f.num;
    int d = this.den * f.den;
    return new Frazione(n, d);
}

public Frazione diviso(Frazione f) {
    int n = this.num * f.den;
    int d = this.den * f.num;
    return new Frazione(n, d);
}

public boolean equals(Frazione f) {
    return this.num == f.num && this.den == f.den;
```

```

}

public boolean isMinore(Frazione f) {
    Frazione g = this.meno(f);
    return g.num < 0;
}

public boolean isMaggiore(Frazione f) {
    Frazione g = this.meno(f);
    return g.num > 0;
}

public String toString() {
    if (den == 0)
        return "impossibile";
    else if (den == 1)
        return String.valueOf(num);
    else
        return num + "/" + den;
}

public int getNumeratore() {
    return num;
}

public int getDenominatore() {
    return den;
}

//METODI STATICI
private static int mcd(int a,int b) {
    int resto;
    do {
        resto = a % b;
        a = b;
        b = resto;
    } while (resto != 0);
    return a;
}
}

```

Svilup
costrui
e minu
La

- I
- C
- C
- P
- P
- A

Preved
in base

7.4 Esempio: la classe Orario

Sviluppiamo ora un semplice esempio di classe, utile per introdurre nuovi concetti. Vogliamo costruire una classe i cui oggetti rappresentino orari. Per brevità ci limiteremo a considerare ore e minuti, ma con modifiche elementari è possibile considerare anche i secondi.²

La classe che vogliamo costruire deve disporre dei seguenti costruttori:

- `public Orario(int hh, int mm)`
Costruisce un oggetto che rappresenta l'orario, in cui le ore sono date dal primo parametro e i minuti dal secondo.
- `public Orario()`
Costruisce un oggetto che rappresenta l'orario attuale, cioè l'orario relativo all'istante in cui viene invocato.
- `public Orario(String s)`
Costruisce un oggetto che rappresenta l'orario indicato nella stringa fornita tramite il parametro.

Prevediamo, inoltre, di fornire i metodi sottoelencati, che per maggiore chiarezza raggruppiamo in base al tipo di funzionalità offerta.

- *Metodi di confronto.*

- `public boolean equals(Orario altro)`
Restituisce `true` quando l'orario rappresentato dall'oggetto che esegue il metodo coincide con quello rappresentato dall'oggetto fornito tramite il parametro.
- `public boolean isMinore(Orario altro)`
Restituisce `true` quando l'orario rappresentato dall'oggetto che esegue il metodo precede quello fornito tramite il parametro, supponendo che i due orari si riferiscano alla stessa giornata. Ad esempio, se l'oggetto che esegue il metodo rappresenta l'orario 23:58 e il parametro l'orario 1:00, il metodo restituisce `false`. Se invece l'oggetto che esegue il metodo rappresenta l'orario 1:00 e il parametro l'orario 23:58, il metodo restituisce `true`.
- `public boolean isMaggiore(Orario altro)`
Restituisce `true` quando l'orario rappresentato dall'oggetto che esegue il metodo segue quello fornito tramite il parametro, supponendo che i due orari si riferiscano alla stessa giornata.

² All'interno del package `java.util`, la libreria standard di Java fornisce diverse classi per la manipolazione di informazioni temporali: in particolare le classi `Date`, `Time`, la classe astratta `Calendar` e la sua sottoclasse concreta `GregorianCalendar`.

- *Metodi di calcolo.*

- `public int quantoManca(Orario altro)`

Restituisce il numero di minuti che intercorrono tra l'orario rappresentato dall'oggetto che esegue il metodo e quello rappresentato dall'oggetto fornito tramite il parametro, considerati come orari riferiti alla stessa giornata. In particolare, se l'orario rappresentato dall'oggetto che esegue il metodo è minore di quello fornito tramite il parametro, il risultato sarà un numero positivo (ad esempio, se i due orari sono rispettivamente 1:00 e 23:58, il risultato sarà 1378); se invece è maggiore, il risultato sarà un numero negativo (se i due orari sono rispettivamente 23:58 e 1:00, il risultato sarà -1378).

- `public String toString()`

Restituisce una stringa che descrive l'orario rappresentato dall'oggetto che esegue il metodo, come "9:04" o "12:55".

- *Metodi di accesso.*

Questi metodi permettono di accedere alle informazioni di base che definiscono un orario (e che, in questo esempio, coincideranno con i campi che utilizzeremo nell'implementazione).

- `public int getOre()`

Restituisce il valore delle ore.

- `public int getMinuti()`

Restituisce il valore dei minuti.

Prima di procedere all'implementazione della classe, sviluppiamo una breve applicazione che ne utilizza alcuni metodi. L'applicazione richiede all'utente di inserire un orario e comunica quanti minuti sono passati dall'orario inserito o quanti minuti mancano.

Le operazioni possono essere schematizzate come segue:

leggi l'orario

calcola l'orario attuale

calcola e comunica la differenza tra i due orari

Più in dettaglio, tenendo conto del fatto che l'orario letto può precedere, seguire o coincidere con l'orario attuale, possiamo scrivere:

leggi l'orario

calcola l'orario attuale

SE l'orario letto è minore di quello attuale

ALLORA

calcola e comunica quanti minuti sono passati dall'orario letto

ALTRIMENTI SE l'orario letto è maggiore di quello attuale

ALLORA

calcola e comunica quanti minuti mancano all'orario letto

```

ALTRIMENTI
    comunica che i due orari coincidono
FINESE
FINESE

```

Passiamo ora alla codifica del metodo `main` dell'applicazione basandoci sullo schema precedente. Prima di tutto creiamo i due oggetti per la comunicazione con l'utente tramite terminale:

```

ConsoleInputManager in = new ConsoleInputManager();
ConsoleOutputManager out = new ConsoleOutputManager();

```

Per la lettura dell'orario utilizziamo una stringa. Questa stringa può essere poi passata al terzo costruttore della classe `Orario` per costruire l'oggetto. Il riferimento all'oggetto sarà memorizzato in una variabile di nome `o` (per semplicità non consideriamo errori che potrebbero derivare dall'inserimento di stringhe in formato scorretto):

```

String s = in.readLine("Inserire un orario (hh:mm) ");
Orario o = new Orario(s);

```

Dobbiamo ora calcolare l'orario attuale e costruire un oggetto che lo rappresenti. A tale scopo è sufficiente servirsi del costruttore senza parametri della classe `Orario` e assegnare il riferimento all'oggetto ottenuto a una variabile che chiamiamo `adesso`:

```
Orario adesso = new Orario();
```

A questo punto dobbiamo codificare il blocco contenente la duplice selezione. Per valutare le due condizioni basta chiedere all'oggetto che rappresenta l'orario inserito dall'utente, cioè all'oggetto riferito da `o`, di eseguire i metodi `isMinore` e `isMaggiore`, fornendo come parametro il riferimento `adesso`.

Qualora l'orario letto sia minore di quello attuale (prima selezione), dobbiamo calcolare e comunicare quanti minuti sono passati dall'orario letto. A tal fine richiediamo all'oggetto riferito da `o` di calcolare quanti minuti mancano all'orario riferito da `adesso`:

```
o.quantoManca(adesso)
```

Il risultato viene inserito direttamente nella stringa da visualizzare:

```

out.println("Dalle " + o.toString() + " di oggi sono passati " +
            o.quantoManca(adesso) + " minuti");

```

Nel caso invece l'orario letto sia maggiore di quello attuale (seconda selezione), si deve comunicare il numero di minuti che mancano all'orario letto. In questo caso si deve dunque chiedere all'oggetto `adesso` di eseguire il metodo `quantoManca` con parametro `o`:

```
adesso.quantoManca(o)
```

Il resto dell'implementazione dell'applicazione può essere compreso senza difficoltà leggendo il testo, che riportiamo qui:

```

import prog.io.*;

class ProvaOrario {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        String s = in.readLine("Inserire un orario (hh:mm) ");
        Orario o = new Orario(s);
        Orario adesso = new Orario();
        out.println("Sono le " + adesso.toString());
        if (o.isMinore(adesso)) {
            out.println("Dalle " + o.toString() + " di oggi sono passati " +
                       o.quantoManca(adesso) + " minuti");
        } else if (o.isMaggiore(adesso)) {
            out.println("Mancano " + adesso.quantoManca(o) + " minuti alle " +
                       o.toString() + " di oggi");
        } else
            out.println("L'ora inserita è quella attuale: " + o.toString());
    }
}

```

Ecco alcuni esempi di esecuzione (supponendo, naturalmente, di disporre già della classe `Orario`):

```

Inserire un orario (hh:mm) 22:30
Sono le 23:50
Dalle 22:30 di oggi sono passati 80 minuti

Inserire un orario (hh:mm) 23:59
Sono le 23:51
Mancano 8 minuti alle 23:59 di oggi

```

Sviluppiamo ora un'implementazione elementare della classe `Orario`. Prima di tutto dobbiamo stabilire come rappresentare lo stato di un oggetto. In altre parole, dobbiamo decidere quali sono le variabili di istanza. Anche in questo caso, la scelta più ovvia consiste nel considerare due campi di tipo `int`, che rappresentano le ore e i minuti. Chiameremo tali campi `ore` e `min`.

Il primo costruttore non deve fare altro che assegnare i valori dei due parametri ai due campi (non effettuiamo alcun controllo sulla correttezza e significatività dei valori inseriti). Pertanto può essere scritto come:

```
public Orario(int hh, int mm) {
    ore = hh;
    min = mm;
}
```

Riportiamo poi il testo degli altri due costruttori senza darne una spiegazione dettagliata, che esula dagli scopi di questo capitolo. In particolare, il costruttore senza argomenti è implementato tramite alcuni metodi e costanti presenti nelle classi `Calendar` e `GregorianCalendar` del package `java.util`, utilizzati per estrarre l'orario attuale. Il costruttore con argomento `String` si aspetta di ricevere una stringa di 5 caratteri: i primi due vengono interpretati come ore, gli ultimi due come minuti. Chiaramente, l'utilizzo di una stringa in formato differente provocherà un errore in esecuzione. Il costruttore potrebbe essere migliorato per riconoscere altri formati.

```
public Orario() {
    GregorianCalendar adesso = new GregorianCalendar();
    ore = adesso.get(Calendar.HOUR_OF_DAY);
    min = adesso.get(Calendar.MINUTE);
}

public Orario(String s) {
    ore = Integer.parseInt(s.substring(0,2));
    min = Integer.parseInt(s.substring(3,5));
}
```

Il metodo `equals` può essere scritto verificando l'uguaglianza nei due orari campo per campo:

```
public boolean equals(Orario altro) {
    return this.ore == altro.ore && this.min == altro.min;
}
```

Vediamo come implementare il metodo `isMinore`. L'orario rappresentato dall'oggetto che esegue il metodo (riferimento `this`) è minore di quello rappresentato dall'oggetto riferito da `altro` se il numero di ore è minore:

```
this.ore < altro.ore
```

oppure se il numero di ore è uguale, ma il numero di minuti è minore:

```
this.ore == altro.ore && this.min < altro.min
```

In tutti gli altri casi l'orario rappresentato dall'oggetto che esegue il metodo non è minore di quello riferito da `altro`. Pertanto il risultato del metodo `isMinore` può essere ottenuto combinando le condizioni precedenti. Riportiamo il testo del metodo, dove, non essendovi ambiguità, possiamo omettere il riferimento `this`:

```
public boolean isMinore(Orario altro) {
    return ore < altro.ore || (ore == altro.ore && min < altro.min);
}
```

Il metodo `quantoManca` può essere codificato calcolando le differenze tra i campi dei due orari, convertendo le ore in minuti:

```
public int quantoManca(Orario altro) {
    return (altro.ore - ore) * 60 + altro.min - min;
}
```

Passiamo ora al metodo `toString`. Una prima codifica si ottiene costruendo una stringa formata dall'indicazione delle ore seguita da un separatore (utilizziamo la stringa ":"), seguita dall'indicazione dei minuti:

```
public String toString() {
    return ore + ":" + min;
}
```

Si noti anzitutto che le due operazioni `+` indicano una concatenazione di stringhe, in quanto uno degli operandi è per entrambe di tipo `String`. Se in luogo di ":" si fosse scritto ":" l'effetto sarebbe stato ben diverso (provate...).

Il risultato fornito dal metodo è insoddisfacente. Se il campo `ore` contiene 9 e il campo `minuti` contiene 10, la stringa restituita è "9:10", ma se ambedue i campi contengono 9, la stringa restituita sarà "9:9" anziché "9:09". Quando il campo `min` contiene un valore minore di 10, bisogna inserire uno zero prima del valore. A tale scopo, prima di calcolare il risultato da restituire, costruiamo una stringa contenente il valore dei minuti, nella quale, se necessario, viene inserito uno zero iniziale. Ecco il nuovo testo del metodo:

```
public String toString() {
    String stringaMin = (min < 10 ? "0" : "") + min;
    return ore + ":" + stringaMin;
}
```

L'implementazione dei metodi restanti è molto semplice. Vediamo il testo completo della classe:

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Orario {
    //CAMPI
    private int ore, min;

    //COSTRUTTORI
    public Orario(int hh, int mm) {
        ore = hh;
        min = mm;
    }
}
```

```
public Orario() {
    GregorianCalendar adesso = new GregorianCalendar();
    ore = adesso.get(Calendar.HOUR_OF_DAY);
    min = adesso.get(Calendar.MINUTE);
}

public Orario(String s) {
    ore = Integer.parseInt(s.substring(0,2));
    min = Integer.parseInt(s.substring(3,5));
}

//METODI
public boolean equals(Orario altro) {
    return this.ore == altro.ore && this.min == altro.min;
}

public boolean isMinore(Orario altro) {
    return ore < altro.ore || (ore == altro.ore && min < altro.min);
}

public boolean isMaggiore(Orario altro) {
    return ore > altro.ore || (ore == altro.ore && min > altro.min);
}

public int quantoManca(Orario altro) {
    return (altro.ore - ore) * 60 + altro.min - min;
}

public String toString() {
    String stringaMin = (min < 10 ? "0" : "") + min;
    return ore + ":" + stringaMin;
}

public int getOre() {
    return ore;
}

public int getMinuti() {
    return min;
}
```

Esercizi

- 7.4 Scrivete una nuova implementazione della classe `Orario` in cui si utilizzi un unico campo di tipo `int` uguale al numero dei minuti trascorsi dalle ore 00:00. Ad esempio l'orario 8:10 sarà rappresentato con il numero 490. Confrontate le implementazioni dei metodi in questa nuova versione con quelle realizzate nel testo: molti metodi risultano più semplici, mentre per altri ci sono alcune complicazioni.

7.5 I campi e i metodi statici

In questo paragrafo riconsideriamo la classe `Orario` sviluppata nel paragrafo precedente introducendo nuove funzionalità utili a illustrare ulteriori concetti.

Prima di tutto vogliamo offrire la possibilità di scegliere un carattere diverso dal “due punti” (`:`) per separare le ore dai minuti nella stringa prodotta dal metodo `toString`. A tale scopo introdurremo un metodo `setSeparatoreTo`, che riceva tramite un parametro di tipo `char` il carattere da utilizzare per separare le ore dai minuti in *tutte* le chiamate successive del metodo `toString` (sino a quando, naturalmente, il separatore non sarà modificato in seguito a una nuova invocazione del metodo `setSeparatoreTo`).

Si osservi che l'effetto desiderato è che il metodo `setSeparatoreTo` stabilisca il separatore da utilizzare per *tutti* gli oggetti di tipo `Orario` (infatti non avrebbe alcun senso utilizzare differenti separatori per orari diversi). Di conseguenza la chiamata del metodo `setSeparatoreTo` non dev'essere rivolta a un singolo oggetto, ma piuttosto all'intera classe. Pertanto il metodo sarà definito statico (si veda la discussione sui metodi statici nel Paragrafo 4.9).

Il carattere da utilizzare come separatore dovrà essere memorizzato in un apposito campo. Tuttavia questo carattere *non definisce* lo stato di un singolo oggetto, ma è un'informazione *comune a tutta la classe*, utilizzata per la rappresentazione sotto forma di stringa degli oggetti. Quest'informazione non deve quindi essere memorizzata all'interno dei singoli oggetti, ma all'interno di un campo comune a tutta la classe.

Il linguaggio Java permette di definire *campi statici*, cioè campi associati all'intera classe e non ai singoli oggetti. I campi statici sono detti anche *variabili di classe*, in contrapposizione alle variabili di istanza, cioè ai campi non statici associati ai singoli oggetti. I campi statici appartengono all'intera classe ed esistono a prescindere dai suoi oggetti. Anche se non è stato istanziato alcun oggetto della classe, i suoi campi statici sono disponibili. Per definire un campo statico è sufficiente utilizzare il modificatore `static` nella definizione del campo.

Per l'esempio che stiamo sviluppando, definiamo nella classe `Orario` un campo statico di tipo `char` e di nome `separatore`. Al fine di nascondere l'implementazione della classe, anche questo campo viene definito `private`. Inoltre assegniamo al campo un valore iniziale, che potrà essere poi modificato utilizzando il metodo `setSeparatoreTo`. A tale scopo basta inserire nel corpo della classe la seguente definizione:

```
private static char separatore = ':';
```

Ora il t

publ

S-

r

}

Esamin

lare es

del tutt

di costi

istanze

classe :

classe.

metodo

publ

se

}

Aggiu

statico:

publ

re

}

Poiché
precede

Orar

Orar

Orar

Orar

out

out

out

out

Orar

out

out

Orar

out

out

³ Si n

interi.

⁴ Oss

classe. N

utilizza i

la leggib

utilizzare

Ora il metodo `toString` può essere riscritto utilizzando la variabile `separatore` come:³

```
public String toString() {
    String stringaMin = (min < 10 ? "0" : "") + min;
    return String.valueOf(ore) + separatore + stringaMin;
}
```

Esaminiamo l'implementazione del metodo `setSeparatoreTo`. Questo metodo deve manipolare *esclusivamente* un'informazione comune a tutta la classe, cioè il campo `separatore`, ed è del tutto indipendente dagli oggetti della classe. Il metodo potrebbe essere invocato anche prima di costruire istanze della classe. Un metodo con queste caratteristiche, svincolato dalle singole istanze della classe, viene dichiarato *statico*. Ricordiamo che un metodo statico è eseguito dalla classe stessa e non dalle sue istanze, e può essere invocato anche se non esistono istanze della classe. Per la dichiarazione è sufficiente utilizzare il modificatore `static` nell'intestazione. Il metodo può essere scritto come:

```
public static void setSeparatoreTo(char ch) {
    separatore = ch;
}
```

Aggiungiamo anche un metodo che restituisca il valore del carattere separatore. Anch'esso sarà statico:

```
public static char getSeparatore() {
    return separatore;
}
```

Poiché un metodo statico è eseguito dalla classe stessa, nell'invocazione il nome del metodo sarà preceduto dal nome della classe, come esemplificato nel seguente frammento di codice:⁴

```
Orario a = new Orario(1,10);
Orario b = new Orario(2,20);
Orario c = new Orario(2,25);
Orario.setSeparatoreTo(',');
out.println(a.toString());
out.println(b.toString());
out.println(c.toString());
Orario.setSeparatoreTo(':');
out.println(a.toString());
```

³ Si noti la conversione del valore di `ore` al tipo `String`. Senza di essa, la prima operazione `+` sarebbe una somma di interi.

⁴ Osserviamo che è consentito invocare un metodo statico anche utilizzando il riferimento a una qualsiasi istanza della classe. Nell'esempio le due invocazioni potrebbero essere scritte come `b.setSeparatoreTo(' ',')`. Questa scrittura utilizza inutilmente il nome di un riferimento per un'invocazione del tutto indipendente dall'oggetto, riducendo quindi la leggibilità del codice. Poiché l'esecuzione di un metodo statico viene richiesta all'intera classe, è sempre opportuno utilizzare il nome di questa; l'uso del riferimento a un oggetto *dev'essere assolutamente evitato*.

```
out.println(b.toString());
out.println(c.toString());
```

Aggiungeremo ora un'ulteriore funzionalità alla classe `Orario`. Mentre in Europa le ore vengono rappresentate con i numeri da 0 a 24, in America si utilizzano di solito i numeri da 1 a 12, con l'indicazione “am” (*ante meridiem*) o “pm” (*post meridiem*), rispettivamente per gli orari che precedono e seguono mezzogiorno. In particolare gli orari da mezzanotte fino a mezzogiorno (escluso) sono seguiti dal suffisso “am”, mentre quelli da mezzogiorno fino a mezzanotte (esclusa) sono seguiti dal suffisso “pm”. Pertanto gli orari “11:59pm”, “11:59am”, “12:01pm” e “12:01am” corrispondono rispettivamente, nella notazione europea, a “23:59”, “11:59”, “12:01” e “0:01”.

Vogliamo modificare la classe `Orario` in modo che il metodo `toString` possa produrre il risultato anche nella notazione americana. Anche in questo caso, la scelta del formato in cui rappresentare gli orari come stringhe non è legata ai singoli oggetti. Introduciamo dunque un campo statico, di tipo `boolean`, che contenga `true` se e solo se dev’essere utilizzata la rappresentazione europea. Ecco la definizione del campo:

```
private static boolean formato24 = true;
```

Segue un metodo `setFormato24` che, ricevendo parametro `false`, disattiva la notazione europea e attiva quella americana, mentre ricevendo parametro `true` attiva la notazione europea:

```
public static void setFormato24(boolean b) {
    formato24 = b;
}
```

Forniamo inoltre un metodo per sapere se la notazione su 24 ore sia attiva:

```
public static boolean isFormato24Attivo() {
    return formato24;
}
```

Per produrre il tipo di rappresentazione selezionata in base al campo statico `formato24`, il metodo `toString` può essere a questo punto riscritto come segue:

```
public String toString() {
    String risultato;
    String stringaMin = (min < 10 ? "0" : "") + min;
    if (formato24)
        risultato = String.valueOf(ore) + separatore + stringaMin;
    else {
        int oraRisultato;
        String suffisso;
        if (ore == 0) {
            oraRisultato = 12;
            suffisso = "am";
        }
        else if (ore > 12) {
            oraRisultato = ore - 12;
            suffisso = "pm";
        }
        else {
            oraRisultato = ore;
            suffisso = "am";
        }
        risultato = String.valueOf(oraRisultato) + ":" + min;
    }
    return risultato;
}
```

```

    } else if (ore > 0 && ore < 12) {
        oraRisultato = ore;
        suffisso = "am";
    } else if (ore == 12) {
        oraRisultato = 12;
        suffisso = "pm";
    } else {
        oraRisultato = ore - 12;
        suffisso = "pm";
    }
    risultato = String.valueOf(oraRisultato) + separatore
        + stringaMin + suffisso;
}
return risultato;
}

```

Riportiamo, infine, il testo dell'intera classe Orario dopo queste modifiche. Sarebbero opportuni ulteriori miglioramenti. In particolare il costruttore con parametro String dovrebbe essere migliorato per ricevere diversi formati, tra cui la notazione americana. Inoltre in questa versione abbiamo totalmente ignorato i problemi di correttezza dei dati. Ad esempio non è detto che una stringa sia interpretabile come orario, o che due interi qualsiasi rappresentino un orario. Utilizzando i concetti e gli strumenti presentati nei capitoli successivi, il lettore potrà migliorare notevolmente questa classe.

```

import java.util.Calendar;
import java.util.GregorianCalendar;

public class Orario {
    //CAMPI STATICI
    private static char separatore = ':';
    private static boolean formato24 = true;

    //CAMPI
    private int ore, min;

    //COSTRUTTORI
    public Orario(int hh, int mm) {
        ore = hh;
        min = mm;
    }

    public Orario() {
        GregorianCalendar adesso = new GregorianCalendar();

```

```

    ore = adesso.get(Calendar.HOUR_OF_DAY);
    min = adesso.get(Calendar.MINUTE);
}

public Orario(String s) {
    ore = Integer.parseInt(s.substring(0,2));
    min = Integer.parseInt(s.substring(3,5));
}

//METODI
public boolean equals(Orario altro) {
    return this.ore == altro.ore && this.min == altro.min;
}

public boolean isMinore(Orario altro) {
    return ore < altro.ore || (ore == altro.ore && min < altro.min);
}

public boolean isMaggiore(Orario altro) {
    return ore > altro.ore || (ore == altro.ore && min > altro.min);
}

public int quantoManca(Orario altro) {
    return (altro.ore - ore) * 60 + altro.min - min;
}

public int getOre() {
    return ore;
}

public int getMinuti() {
    return min;
}

public String toString() {
    String risultato;
    String stringaMin = (min < 10 ? "0" : "") + min;
    if (formato24)
        risultato = String.valueOf(ore) + separatore + stringaMin;
    else {
        int oraRisultato;
        String suffisso;
}

```

```

    if (ore == 0) {
        oraRisultato = 12;
        suffisso = "am";
    } else if (ore > 0 && ore < 12) {
        oraRisultato = ore;
        suffisso = "am";
    } else if (ore == 12) {
        oraRisultato = 12;
        suffisso = "pm";
    } else {
        oraRisultato = ore - 12;
        suffisso = "pm";
    }
    risultato = String.valueOf(oraRisultato) + separatore
        + stringaMin + suffisso;
}
return risultato;
}

//METODI STATICI
public static void setFormato24(boolean b) {
    formato24 = b;
}

public static void setSeparatoreTo(char ch) {
    separatore = ch;
}

public static boolean isFormato24Attivo() {
    return formato24;
}

public static char getSeparatore() {
    return separatore;
}
}

```

Esercizi

- 7.5 Implementate una classe `Ora`, analoga a `Orario`, in cui si considerino anche i secondi. La classe deve contenere gli stessi metodi della classe `Orario` e costruttori con la stessa lista di

parametri di quelli forniti dalla classe `Orario`, più gli altri metodi e costruttori che riterrete opportuni.

7.6 Scrivete un'implementazione della classe `Data` definita nel package `prog.utili` (per ora non preoccupatevi di implementare l'interfaccia `Comparable`).

7.7 Scrivete un'implementazione della classe `Importo` definita nel package `prog.utili` (per ora non preoccupatevi di implementare l'interfaccia `Comparable`).

Si sconsiglia l'utilizzo del tipo `double` per rappresentare all'interno della classe l'importo considerato. Si può utilizzare invece un unico campo `int` in cui la somma viene rappresentata in centesimi di Euro (ad esempio la quantità 123.45 sarà rappresentata con l'intero 12345), oppure si possono utilizzare due campi `int`, uno per la parte intera e uno per i centesimi. In ogni caso fate attenzione alle questioni legate agli arrotondamenti.

7.8 Scrivete una classe di prova per i metodi della classe `Importo`.

7.9 Aggiungete alla classe `Importo` un nuovo metodo che fornisca le somme in lettere:

- `public String toLetterString()`

Fornisce una stringa contenente l'indicazione in lettere della somma rappresentata dall'oggetto che esegue il metodo, secondo il formato convenzionalmente utilizzato per scrivere gli assegni, in cui la parte intera è espressa in lettere, mentre i centesimi sono scritti in cifre. Ad esempio le somme 123.45 e 20 vengono scritte come `centoventitre/45 e venti/00`.

7.10 Aggiungete alla classe `Orario` un metodo statico `parseOrario`, che consenta di convertire una stringa in un `Orario` tenendo conto del separatore utilizzato.

7.6 Riepilogo della struttura delle classi

In questo paragrafo riassumiamo gli elementi che abbiamo introdotto finora nel capitolo. Anzi-tutto possiamo dire che il corpo di una classe è costituito da:

- campi
- campi statici
- costruttori
- metodi
- metodi statici.

Tutti questi elementi sono anche detti *membri* della classe (da non confondere con gli oggetti, che sono *istanze* della classe). Sintetizziamo alcuni elementi relativi ai membri di una classe.

Campi

I campi, detti anche *variabili di istanza*, sono associati a ciascun oggetto della classe e ne definiscono lo *stato*. Al momento della creazione di un oggetto della classe si riserva all'interno di esso lo spazio di memoria necessario a memorizzare i campi. I dati memorizzati in queste variabili vivono insieme con l'oggetto: finché l'oggetto è accessibile, essi sono presenti, anche se l'oggetto non sta eseguendo alcun metodo.

Per accedere a un campo si utilizza la sintassi:

riferimento_a_oggetto.nome_campo

Per riferirsi all'oggetto in esecuzione all'interno del codice della classe si può utilizzare il riferimento **this**. Quando si sottintenda l'oggetto in esecuzione tale riferimento può essere omesso; in questo caso è sufficiente scrivere il nome del campo.

Un campo dichiarato **private** non è accessibile all'esterno del codice della classe.

I campi vengono automaticamente inizializzati dalla Java Virtual Machine al momento della creazione dell'oggetto; il valore utilizzato per l'inizializzazione dipende dal tipo del campo ed è 0 per i campi di tipo intero, 0.0 per i campi di tipo **float** e **double**, **false** per i campi di tipo **boolean**, '\u0000' per i campi di tipo **char**, e **null** per i campi di tipo riferimento.

Nella dichiarazione dei campi è possibile ovviamente indicare un valore iniziale. Ad esempio, nel seguente frammento della classe A

```
public class A {
    public int x = 1;
    private double y;
    boolean b;
    ...
}
```

risultano dichiarati tre campi: un campo **x** di tipo **int** che è esplicitamente inizializzato a 1, un campo **y** di tipo **double** che, mancando un'inizializzazione esplicita, viene inizializzato dalla Java Virtual Machine a 0.0, e un campo **b** di tipo **boolean**, anch'esso mancante di un'inizializzazione esplicita e quindi inizializzato dalla Java Virtual Machine a **false**. Inoltre i campi **x** e **b** sono visibili (e quindi utilizzabili) al di fuori del codice della classe, in quanto **x** è dichiarato **public** mentre **b** non ha modificatori di visibilità (vedremo la differenza fra la dichiarazione **public** e quella senza modificatori quando discuteremo la definizione dei package). Al contrario il campo **y** non è visibile (e quindi non è utilizzabile) al di fuori del codice della classe, in quanto dichiarato **private**.

Campi statici

I campi statici contengono informazioni comuni a tutta la classe; sono creati dalla Java Virtual Machine quando viene eseguita la prima istruzione che contiene il nome della classe, ed esistono indipendentemente dagli oggetti della classe stessa.

I campi statici sono definiti facendo precedere la dichiarazione della variabile dal modificatore `static` e, come i campi non statici, vengono inizializzati automaticamente al valore di default dalla Java Virtual Machine al momento della creazione. Si può assegnare un valore all'atto della dichiarazione anche ai campi statici. Consideriamo il seguente frammento di una classe B:

```
public class B {
    static int contaOggetti = 0;
    private int x;

    public B (int i) {
        contaOggetti++;
        x = i;
    }
    ...
}
```

Mentre `x` è un campo, `contaOggetti` è un campo statico. Nell'esempio, `contaOggetti` è inizializzato esplicitamente a 0 (lo stesso valore cui sarebbe stato inizializzato di default se non avessimo fornito un'inizializzazione esplicita). Lo scopo di `contaOggetti` nella classe B è quello di memorizzare il numero delle istanze dell'oggetto costruite; esso viene infatti incrementato all'interno del costruttore della classe. Osserviamo che all'interno del codice della classe è possibile accedere al campo statico per mezzo del solo nome. Al contrario, per utilizzare un campo statico al di fuori della classe si deve scrivere:

nome_classe.nome_campo_statico

Ad esempio, tramite `B.contaOggetti` è possibile accedere al campo statico della classe B (ovviamente è possibile accedere al campo statico mediante `B.contaOggetti` anche all'interno del codice della classe). Anche nel caso dei campi statici si può limitarne la visibilità all'interno della classe; ad esempio, se avessimo usato in B la dichiarazione:

```
private static int contaOggetti = 0;
```

il campo `contaOggetti` sarebbe sempre stato accessibile all'interno della classe, ma non al di fuori di essa.

Costruttori

Un *costruttore* è una porzione di codice che ha lo stesso nome della classe e viene utilizzata per creare un nuovo oggetto della classe. In genere il codice del costruttore viene usato per inizializzare le variabili di istanza dell'oggetto creato. Il costruttore ha una propria lista di parametri (come i metodi) ed è chiamato in un'espressione `new`, il cui risultato è un riferimento all'oggetto creato.

Metodi e metodi statici

Anche i metodi si dividono in statici e non statici. I metodi statici appartengono all'intera classe. Per invocarli è necessario far precedere il nome del metodo dal nome della classe. I metodi non statici appartengono invece ai singoli oggetti. Per invocare un metodo non statico occorre far precedere il nome del metodo da un riferimento all'oggetto, secondo le stesse modalità indicate per i campi.

Supponiamo, ad esempio, che il seguente metodo statico appartenga alla classe B:

```
public static void azzera() {
    contaOggetti = 0;
}
```

Mediante B.azzera() si esegue il metodo azzera() della classe B. Sottolineiamo che mentre i metodi non statici possono accedere anche ai membri statici della classe non è possibile il contrario. Che senso avrebbe infatti per un metodo statico, ossia eseguito dall'intera classe, accedere a un campo non statico, cioè a un campo presente in ogni istanza della classe? Al campo di quale istanza dovrebbe accedere il metodo? Si noti inoltre che campi e metodi statici possono essere utilizzati anche se non sono state create istanze della classe. Ad esempio, possiamo aggiungere alla classe B il metodo (non statico):

```
public String toString() {
    return "x = " + x + ", " + "numero istanze = " + contaOggetti;
}
```

che utilizza il campo statico, ma non possiamo definire un metodo statico che utilizzi il campo x.

Variabili locali e parametri

Introducendo metodi e costruttori abbiamo iniziato a evidenziare il ruolo di variabili locali e parametri. A questo punto vediamo di riassumerne le principali caratteristiche.

Le variabili dichiarate all'interno di un metodo o di un costruttore prendono il nome di *variabili locali*. Le variabili dichiarate nell'intestazione del metodo o del costruttore prendono il nome di *parametri formali*. Ad esempio, nel metodo:

```
public boolean equals(Frazione f) {
    Frazione g = this.meno(f);
    if (g.num == 0)
        return true;
    else
        return false;
}
```

della classe Frazione, f è un parametro formale, mentre g è una variabile locale. Le variabili dichiarate localmente a un metodo e i parametri di questo vengono creati dinamicamente al momento dell'esecuzione del metodo stesso e distrutti alla sua conclusione (rientro dal metodo).

A differenza di quanto accade con i campi (statici e non statici), le variabili locali dei metodi non hanno un'inizializzazione automatica; i parametri sono invece inizializzati al momento della chiamata, con i rispettivi argomenti indicati nell'invocazione del metodo.

Uso di this

La parola riservata `this` ha un doppio significato, a seconda del contesto in cui compare. Se è utilizzata all'interno di un costruttore seguita da una parentesi tonda aperta, da una lista di argomenti e da una parentesi tonda chiusa, è interpretata dal compilatore come l'invocazione di un altro costruttore della stessa classe. In questa accezione di costruttore, `this` può essere utilizzata solo come prima istruzione del corpo del costruttore.

Oltre a questo significato, `this` può essere usata come una variabile riferimento all'interno di un metodo o di un costruttore in relazione all'oggetto corrente, cioè all'oggetto che sta eseguendo il metodo. In realtà `this` è una *pseudo-variabile*, nel senso che durante l'esecuzione il suo valore cambia (come quello di ogni altra variabile), ma sotto il controllo della Java Virtual Machine e non del programmatore, che non può modificarla. Ad esempio, un assegnamento del tipo

```
this = ...;
```

non viene accettato dal compilatore. Un riferimento `this` è implicito ogni volta che, all'interno di un metodo o di un costruttore, si fa riferimento a un campo o a un metodo della classe stessa. Ad esempio il metodo `equals` della prima versione della classe `Frazione` (riportato sopra) potrebbe essere riscritto evitando l'uso di `this` nell'invocazione del metodo `meno`:

```
public boolean equals(Frazione f) {
    Frazione g = meno(f);
    if (g.num == 0)
        return true;
    else
        return false;
}
```

La scelta di utilizzare o no `this` in questi casi è essenzialmente legata a ragioni stilistiche. Spesso l'uso di `this` migliora la leggibilità del codice, altre volte risulta superfluo. Ovviamente esistono casi in cui `this` ha invece un ruolo determinante: ne discuteremo quando parleremo dell'adombramento.

Il garbage collector

Abbiamo detto in precedenza che quando viene creato un oggetto, la Java Virtual Machine gli riserva uno spazio in memoria. Naturalmente, essendo la memoria disponibile limitata, è bene che quando un oggetto non esiste più la Java Virtual Machine recuperi lo spazio di memoria che esso occupava. Java non prevede un'istruzione per "distruggere" esplicitamente un oggetto, ma considera un oggetto "eliminabile" quando non esistono più riferimenti a esso all'interno dell'applicazione in esecuzione. È evidente infatti che se in un'applicazione non esiste alcun riferimento

a un oggetto, l'oggetto non è più accessibile, ossia non esiste alcun modo di utilizzarlo. Quando un oggetto non è più accessibile, la sua area di memoria può essere recuperata e riutilizzata per altri scopi. Il *garbage collector* (raccoglitore di rifiuti) è una routine della Java Virtual Machine che individua gli oggetti non più accessibili, recuperandone gli spazi di memoria.

7.7 Implementazione di un'interfaccia

Come abbiamo spiegato nel Paragrafo 6.12, le interfacce Java permettono di specificare un insieme di *comportamenti* senza però fornirne l'implementazione. Le interfacce specificano cioè il prototipo di alcuni metodi, che sono a tutti gli effetti metodi astratti, e il loro contratto, ma non ne forniscono l'implementazione, rimandandola alle *classi che implementano l'interfaccia*. In questo paragrafo vedremo come implementare un'interfaccia in una classe mostrando l'implementazione dell'interfaccia generica `Comparable` da parte della classe `Frazione`.

Per prima cosa torniamo a considerare l'interfaccia `Comparable`. Si tratta di un'interfaccia generica che prevede un tipo parametro `T` e specifica un solo metodo così descritto:

- `public int compareTo(T o)`

Confronta l'oggetto che esegue il metodo con quello specificato come argomento, e restituisce un intero negativo, zero, o un intero positivo, a seconda che l'oggetto che esegue il metodo sia minore, uguale o maggiore di quello specificato come argomento.

Affinché una classe *implementi* un'interfaccia è necessario:

- dichiarare nell'intestazione della classe (utilizzando la parola chiave `implements`) che questa implementa l'interfaccia;
- implementare, cioè definire, all'interno della classe i metodi dichiarati nell'interfaccia.

Affinché la classe `Frazione` implementi l'interfaccia `Comparable` dovremo quindi dichiarare che essa implementa l'interfaccia nel modo seguente:

```
public class Frazione implements Comparable<Frazione> {
    ...
}
```

Si noti che, essendo `Comparable` generica, viene fornito un tipo argomento, in questo caso `Frazione`. In questo modo il metodo `compareTo` previsto dall'interfaccia avrà un parametro di tipo `Frazione` e dunque permetterà di confrontare la frazione che lo esegue con quella fornita tramite l'argomento. Il metodo può essere implementato senza difficoltà tramite i metodi `isMinore` ed `equals` della classe `Frazione`:

```
public int compareTo(Frazione altra) {
    if (this.equals(altra))
        return 0;
    else
```

```

    if (this.isMinore(altra))
        return -1;
    else
        return 1;
}

```

Con l'aggiunta di questo metodo, l'implementazione dell'interfaccia Java da parte della classe **Frazione** è completa.

Si noti che i metodi dell'interfaccia devono essere implementati con *lo stesso prototipo* specificato nell'interfaccia, questo significa che l'implementazione deve rispettare i modificatori di visibilità (tutti i metodi definiti in un'interfaccia sono `public`) e la segnatura. Sono invece influenti i nomi dei parametri formali, che possono essere adattati per rendere più chiaro il loro significato.

Diversamente da quanto accade nel caso dell'estensione delle classi, è possibile far implementare a una classe più interfacce. La struttura generale dell'intestazione di una classe è quindi la seguente:

```

public class Nome extends ClasseBase implements Interfaccia1, ..., InterfacciaN {
    ...
}

```

Se nell'intestazione di una classe si dichiara che sarà implementata un'interfaccia, ma nel corpo della classe non si definiscono i suoi metodi, il compilatore segnala un errore. Ecco il messaggio fornito compilando la classe **Frazione**, se si dichiara che essa implementa l'interfaccia `Comparable<Frazione>` ma non si definisce il metodo `compareTo`:

```

> javac Frazione.java
Frazione.java:1: Frazione is not abstract and does not override abstract
method compareTo(Frazione) in java.lang.Comparable
public class Frazione implements Comparable<Frazione> {
    ^
1 error

```

Il compilatore segnala che la classe **Frazione** non è stata dichiarata *astratta*. Afferendo nell'intestazione che **Frazione** implementa l'interfaccia `Comparable` stiamo promettendo di fornirne un'implementazione; se non lo facciamo, o stiamo sbagliando oppure intendiamo aggiungere solo il prototipo del metodo `compareTo` alla classe senza fornirne un'implementazione, cioè vogliamo aggiungere alla classe un metodo *abstract*. In questo caso il compilatore pretende che si specifichi che la classe è astratta.

Esercizi

- 7.11 Modificate la classe `Orario` descritta nel Paragrafo 7.4 e le classi `Data` e `Importo` sviluppate per gli Esercizi 7.6 e 7.7 in modo che implementino l'interfaccia `Comparable`.

- 7.12 Utilizzando le classi `Orario`, `Data` e `SequenzaOrdinata` del package `prog.utili`, scrivete un'applicazione di nome `Agenda` che legga da un file di testo una sequenza di appuntamenti nel formato:

```
gg.mm.aaaa mm:hh descrizione
```

e visualizzi la sequenza degli appuntamenti ordinati per data, ora e descrizione. Si osservi che un singolo appuntamento dovrebbe essere modellato da una classe `Appuntamento` che conservi nel suo stato le informazioni relative a esso. Inoltre, per poter gestire la sequenza degli appuntamenti mediante una sequenza ordinata è necessario che la classe `Appuntamento` implementi l'interfaccia `Comparable`.

- 7.13 Modificate l'applicazione precedente in modo che permetta di inserire nella sequenza nuovi appuntamenti e permetta di salvarne su file la sequenza.

7.8 Documentazione delle classi

Come abbiamo visto nella prima parte del testo, per poter utilizzare classi già definite è fondamentale disporre di una documentazione precisa delle stesse. Se osserviamo la documentazione delle API (*Application Programmer Interface*), possiamo notare che alcune informazioni contenute in essa sono immediatamente deducibili dal codice, ad esempio il nome dei metodi e il loro prototipo; altre informazioni, come il contratto dei metodi e delle classi, non possono invece esserne ricavate direttamente.

Uno dei pregi del linguaggio Java è quello di aver previsto un meccanismo per la documentazione delle classi integrato con il codice stesso, che permette di generare una documentazione completa delle classi a partire da informazioni già contenute nel codice e da appositi commenti specificati da chi ha implementato la classe. La documentazione viene quindi generata in formato HTML utilizzando un apposito strumento di nome `javadoc`, fornito con l'ambiente di sviluppo.

Consideriamo ad esempio la classe `Frazione` che abbiamo sviluppato nel paragrafo precedente. Come abbiamo detto, il codice della classe contiene già informazioni indispensabili per la documentazione di questa. Le informazioni possono essere estratte mediante `javadoc` impartendo il comando

```
> javadoc Frazione.java
```

In questo caso vengono generati i file relativi alla documentazione di tale classe, tra cui il file `index.html`, che fornisce un punto di ingresso alla documentazione, e il file `Frazione.html`, che contiene la documentazione vera e propria della classe (questi file possono essere visualizzati mediante un qualunque browser HTML). In questo caso la documentazione riporta però solo le informazioni che `javadoc` è in grado di estrarre direttamente dal codice; ad esempio, del metodo `isMaggiore` è riportato esclusivamente il prototipo.

Le informazioni necessarie a completare la documentazione della classe possono essere specificate mediante i *commenti di documentazione*. Si tratta di commenti che iniziano con la sequenza di caratteri `/**` e terminano con la sequenza di caratteri `*/`. Ogni commento di documentazione si riferisce all'identificatore la cui dichiarazione segue il commento. Ad esempio nel codice della classe `Frazione` potremmo specificare il contratto della classe inserendo un commento di documentazione prima della definizione dell'identificatore `Frazione` (cioè prima dell'intestazione della classe), in questo modo:

```
/**  
 * Un oggetto della classe <code>Frazione</code> rappresenta una  
 * frazione.  
 */  
public class Frazione implements Comparable<Frazione> {  
    ...  
}
```

Analogamente, possiamo specificare il contratto del metodo `isMaggiore` inserendo un commento di documentazione prima della definizione del metodo stesso, nel modo seguente:

```
/**  
 * Confronta la frazione rappresentata dall'oggetto che esegue il  
 * metodo con la frazione specificata come argomento, restituisce  
 * <code>true</code> se la frazione che esegue il metodo è maggiore  
 * di quella specificata come argomento, <code>false</code>  
 * altrimenti.  
 */  
public boolean isMaggiore(Frazione f) {  
    Frazione g = this.meno(f);  
    return (g.num > 0);  
}
```

All'interno dei commenti di documentazione è possibile utilizzarc dei *marcatori* (*tag*) HTML come direttive di formattazione (ad eccezione dei marcatori `<h1>`, `<h2>`, ... che sono riservati a javadoc). Ad esempio nei commenti di documentazione descritti sopra abbiamo utilizzato la coppia di marcatori `<code>...</code>`, i quali indicano che la stringa compresa fra loro dev'essere interpretata come un frammento di codice; di solito queste informazioni vengono visualizzate con un font monospaziato.

I commenti di documentazione possono inoltre contenere marcatori che specificano comandi di javadoc. Tutti i marcatori di javadoc iniziano con il carattere @, come ad esempio `@parameter` e `@return`, che sono utilizzati per specificare la documentazione dei parametri e del valore restituito da un metodo. Ad esempio la documentazione completa del metodo `isMaggiore` deve contenere una descrizione del significato dei parametri e del valore restituito dal metodo, che può essere specificata nel modo seguente:

```
/**  
 * Confronta la frazione rappresentata dall'oggetto che esegue il
```

```

* metodo con la frazione specificata come argomento, restituisce
* <code>true</code> se la frazione che esegue il metodo è maggiore
* di quella specificata come argomento, <code>false</code>
* altrimenti.
* @param f la frazione da confrontare con quella che esegue il
*         metodo.
* @return <code>true</code> se la frazione che esegue il metodo è
*         maggiore di quella specificata come argomento,
*         <code>false</code> altrimenti.
*/
public boolean isMaggiore(Frazione f) {
    Frazione g = this.meno(f);
    return (g.num > 0);
}

```

Presentiamo ora alcuni marcatori di javadoc comunemente utilizzati nella documentazione delle classi e dei metodi; per una descrizione di tutti i marcatori disponibili si faccia riferimento alla documentazione di javadoc.

Marcatori per la documentazione delle classi

- **@author *autore***

Specifica il nome dell'autore del codice, indicato dalla stringa *autore*. È possibile specificare più di un autore utilizzando più marcatori **@author**, uno per linea. L'informazione sugli autori è inclusa nella documentazione solo se javadoc viene utilizzato con l'opzione **-author**.

- **@version *versione***

Specifica la versione del codice, indicata dalla stringa *versione*. L'informazione sulla versione è inclusa nella documentazione solo se javadoc viene utilizzato con l'opzione **-version**.

Marcatori per la documentazione dei metodi e dei costruttori

- **@param *nome_parametro* *descrizione***

Viene utilizzato per documentare i parametri di un metodo o di un costruttore. *nome_parametro* è uno degli identificatori che compaiono nella lista dei parametri del metodo e *descrizione* è un testo che lo descrive. Il testo può estendersi su più righe successive.

- **@return *descrizione***

Viene utilizzato per descrivere il valore restituito da un metodo. *descrizione* è un testo che descrive il valore restituito dal metodo.

- `@deprecated`

Viene utilizzato per indicare membri (in genere metodi e costruttori) della classe che sono stati affiancati da una versione migliore e che quindi in futuro potrebbero essere eliminati. Per questa ragione non dovrebbero essere più utilizzati. Il compilatore impiegato con l'opzione `-deprecation` segnala i membri dichiarati `@deprecated` utilizzati all'interno del codice compilato.

In conclusione di questo paragrafo presentiamo a titolo d'esempio il codice completo, comprensivo di documentazione, della classe `Frazione`.

```
/**
 * Un oggetto della classe <code>Frazione</code> rappresenta una
 * frazione.
 * @author Giovanni Pighizzini
 * @author Mauro Ferrari
 * @version 3.0
 */
public class Frazione implements Comparable<Frazione> {
    // CAMPI
    private int num; // il numeratore della frazione
    private int den; // il denominatore della frazione

    // COSTRUTTORI
    /**
     * Costruisce una nuova frazione il cui valore è il rapporto fra il
     * primo argomento e il secondo argomento.
     * @param x numeratore.
     * @param y denominatore.
     */
    public Frazione(int x, int y) {
        if (y == 0) {
            num = 0;
            den = 0;
        } else {
            // memorizza il segno
            boolean negativo = (x < 0 && y > 0) || (x > 0 && y < 0);
            if (x < 0)
                x = -x; // elimina l'eventuale segno meno davanti a x
            if (y < 0)
                y = -y; // elimina l'eventuale segno meno davanti a y

            int m = mcd(x, y);
            if (negativo)
```

```
        num = -x / m; // il segno viene memorizzato al numeratore
    else
        num = x / m;
    den = y / m;
}
}


```

```
    return new Frazione(n, d);
}

/** 
 * Restituisce il riferimento a un nuovo oggetto che rappresenta la
 * frazione ottenuta moltiplicando la frazione specificata come
 * argomento per quella che esegue il metodo.
 * @param f la frazione da moltiplicare per quella che esegue il
 *          metodo.
 * @return la frazione ottenuta moltiplicando quella che esegue il
 *         metodo per quella specificata come argomento.
 */
public Frazione per(Frazione f) {
    int n = this.num * f.num;
    int d = this.den * f.den;
    return new Frazione(n, d);
}

/** 
 * Restituisce il riferimento a un nuovo oggetto che rappresenta la
 * frazione ottenuta dividendo la frazione che esegue il metodo per
 * quella specificata come argomento.
 * @param f la frazione per cui dividere quella che esegue il metodo.
 * @return la frazione ottenuta dividendo quella che esegue il metodo
 *         per quella specificata come argomento.
 */
public Frazione diviso(Frazione f) {
    int n = this.num * f.den;
    int d = this.den * f.num;
    return new Frazione(n, d);
}

/** 
 * Confronta la frazione rappresentata dall'oggetto che esegue il
 * metodo con la frazione specificata come argomento. Restituisce
 * <code>true</code> se le due frazioni hanno lo stesso valore.
 * @param f la frazione da confrontare con quella che esegue il
 *          metodo.
 * @return <code>true</code> se la frazione che esegue il metodo è
 *         uguale a quella specificata come argomento,
 *         <code>false</code> altrimenti.
 */

```

```
public boolean equals(Frazione f) {
    return this.num == f.num && this.den == f.den;
}

/*
 * Confronta la frazione rappresentata dall'oggetto che esegue il
 * metodo con la frazione specificata come argomento, restituisce
 * <code>true</code> se la frazione che esegue il metodo è minore
 * di quella specificata come argomento, <code>false</code>
 * altrimenti.
 * @param f la frazione da confrontare con quella che esegue il
 *          metodo.
 * @return <code>true</code> se la frazione che esegue il metodo è
 *         minore di quella specificata come argomento,
 *         <code>false</code> altrimenti.
 */
public boolean isMinore(Frazione f) {
    Frazione g = this.meno(f);
    return (g.num < 0);
}

/*
 * Confronta la frazione rappresentata dall'oggetto che esegue il
 * metodo con la frazione specificata come argomento, restituisce
 * <code>true</code> se la frazione che esegue il metodo è maggiore
 * di quella specificata come argomento, <code>false</code>
 * altrimenti.
 * @param f la frazione da confrontare con quella che esegue il
 *          metodo.
 * @return <code>true</code> se la frazione che esegue il metodo è
 *         maggiore di quella specificata come argomento,
 *         <code>false</code> altrimenti.
 */
public boolean isMaggiore(Frazione f) {
    Frazione g = this.meno(f);
    return (g.num > 0);
}

/*
 * Restituisce una stringa di caratteri che descrive la frazione
 * rappresentata dall'oggetto che esegue il metodo.
 * @return la stringa che rappresenta la frazione.
```

```

/*
public String toString() {
    if (den == 0)
        return "impossibile";
    else if (den == 1)
        return String.valueOf(num);
    else
        return num + "/" + den;
}
//*
pr

/** Per o
 * Restituisce il valore del numeratore della frazione.
 * @return il valore del numeratore della frazione che esegue il ja
 *         metodo.
 */
public int getNumeratore() {
    return num;
}

/** Si oss
 * Restituisce il valore del denominatore della frazione.
 * @return il valore del denominatore della frazione che esegue il che si
 *         metodo.
 */
public int getDenominatore() {
    return den;
}

// IMPLEMENTAZIONE DI Comparable
/** 7.14
 * Confronta la frazione che esegue il metodo con quella specificata
 * come argomento e restituisce un intero negativo, zero, o un intero
 * positivo a seconda che la <code>Frazione</code> che esegue il metodo
 * sia minore, uguale o maggiore di quella specificata come argomento.
 * @param altra la frazione da confrontare con quella che esegue il
 *             metodo.
 * @return un intero negativo, zero, o positivo a seconda che la
 *         frazione che esegue il metodo sia minore, uguale o
 *         maggiore di quella specificata come argomento.
 */
public int compareTo(Frazione altra) {
    if (this.equals(altra))

```

}

ja

Si oss

ché si

imple

Ese

7.14

7.9

I pack
escimp
packa
If
sappia
l'este
interfa

```

        return 0;
    else if (this.isMinore(altra))
        return -1;
    else
        return 1;
}

// METODI STATICI
/** Calcola il massimo comun divisore fra due numeri. */
private static int mcd(int a, int b) {
    int resto;
    do {
        resto = a % b;
        a = b;
        b = resto;
    } while (resto != 0);
    return a;
}
}

```

Per ottenere la documentazione completa si deve utilizzare il comando:

```
javadoc -author -version Frazione.java
```

Si osservi che nella documentazione prodotta non compaiono i metodi e i campi privati perché si tratta di membri che non sono utilizzabili al di fuori della classe e costituiscono dettagli implementativi della classe stessa.

Esercizi

- 7.14 Scrivete la documentazione della classe `Orario` utilizzando i commenti di documentazione e generatela utilizzando il comando `javadoc`.

7.9 I package

I package permettono di raggruppare in unità logiche classi e interfacce correlate tra loro; ad esempio le classi che costituiscono un'applicazione possono essere raggruppate in uno o più package.

I file che contengono codice sorgente Java vengono chiamati *unità di compilazione*. Come sappiamo, per essere compilata ogni unità di compilazione deve avere un nome che termina con l'estensione `.java`. Di solito si definisce un'unità di compilazione per ogni classe e per ogni interfaccia. Questo però non è indispensabile: infatti è possibile raggruppare in un'unica unità

di compilazione diverse classi e interfacce; il solo vincolo è che ogni unità di compilazione contenga al massimo una classe o un'interfaccia dichiarata `public` e che tale classe o interfaccia abbia il medesimo nome del file.

Per aggiungere una classe o un'interfaccia a un package è necessario dichiararne l'appartenenza mediante l'istruzione

```
package nome_package;
```

Tale istruzione deve obbligatoriamente essere la prima a comparire nell'unità di compilazione (quindi può essere preceduta solo da linee vuote e da commenti). Se l'unità di compilazione contiene più classi e/o interfacce, saranno tutte incluse nel package.

Consideriamo come esempio il package `prog.utili` fornito con il testo. Per aggiungere a esso le classi `Frazione` e `Orario` dobbiamo anzitutto inserire l'istruzione

```
package prog.utili;
```

all'inizio dei file `Frazione.java` e `Orario.java`.

Come indicato nel Capitolo 2, nel nome di un package il punto indica il separatore fra le directory. Pertanto, quando attribuiamo il nome `prog.utili` al package, sottintendiamo che i file contenenti il bytecode delle classi e delle interfacce che lo compongono si trovano, nel caso di sistemi Unix o Linux, in una directory di nome `prog/utilis/`. Inoltre, per poter utilizzare le classi del package dovremo porre tale sottodirectory in una delle directory specificate nella variabile di sistema `CLASSPATH`. Se ad esempio la sequenza delle directory elencate nella variabile di sistema `CLASSPATH` include solo la directory

```
/myjavalib
```

dovremo posizionare la directory `prog/utilis` a partire dalla directory `/myjavalib`, creando quindi nel file system la struttura di directory descritta nella Figura 7.2.

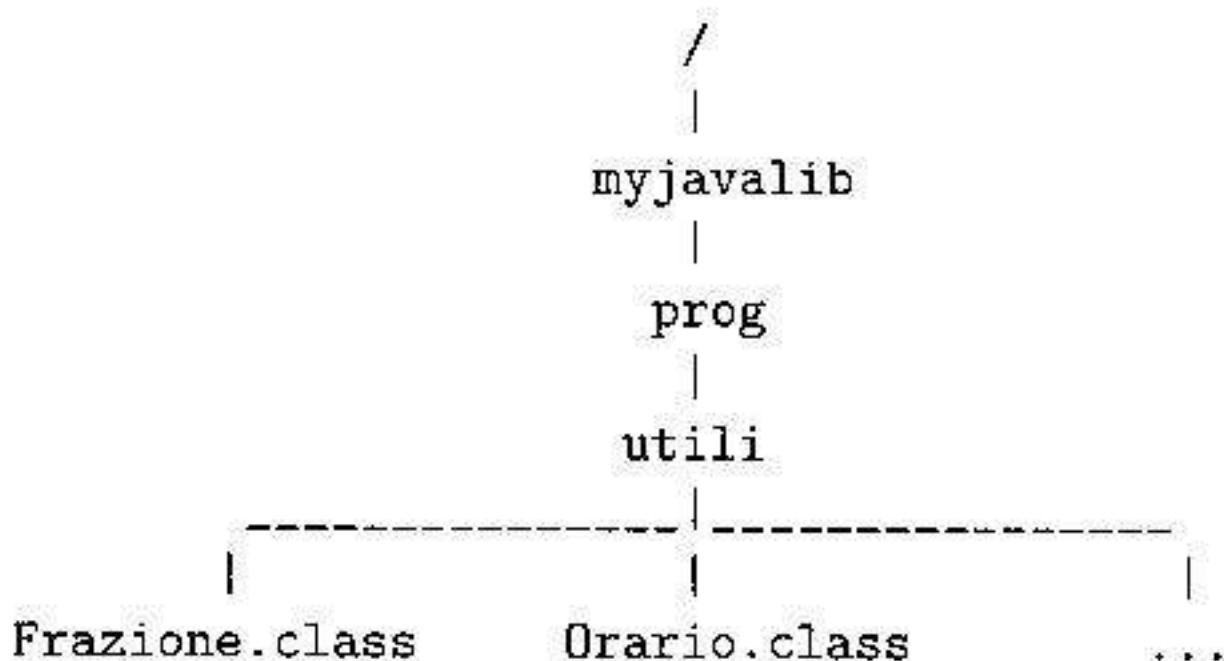


Figura 7.2 Struttura della directory corrispondente al package `prog.utili`.

Per comprendere meglio l'effetto della creazione di un package, costruiamo un package di prova. Per prima cosa creiamo una directory di nome `pack1` in uno dei punti specificati nella variabile di sistema `CLASSPATH` e poniamo in questa directory il file `A.java` contenente la classe:

```
package pack1;

public class A {
    public String toString() {
        return "classe pack1.A";
    }
}
```

A questo punto, possiamo compilare la classe all'interno di questa directory con il comando

```
> javac A.java
```

Dato che la classe è sintatticamente corretta, sarà compilata producendo il file `A.class` nella stessa directory. Diversamente da quanto accade compilando una classe che non è stata inclusa in un package, come abbiamo fatto finora ogni volta che abbiamo definito una nuova classe, il *nome completo* della classe è dato dalla combinazione del nome del package e del nome della classe. Quindi nel nostro esempio, il nome completo della classe `A` è `pack1.A`. Analogamente, il nome completo della classe `ConsoleOutputManager` è `prog.io.ConsoleOutputManager` e il nome completo della classe `String` è `java.lang.String`.

Si può utilizzare una classe indicandone il nome completo e omettendo la direttiva di importazione, come esemplificato dalla seguente applicazione:

```
class Prova {
    public static void main(String[] args) {
        prog.io.ConsoleOutputManager out =
            new prog.io.ConsoleOutputManager();

        pack1.A a = new pack1.A();
        out.println(a.toString());
    }
}
```

Le direttive di importazione non fanno altro che aggiungere lo *spazio dei nomi* di un package, cioè i nomi delle classi e delle interfacce del package, a quanto il compilatore prende in considerazione. Ad esempio la direttiva di importazione `import pack1.A` aggiunge allo spazio dei nomi utilizzato dal compilatore il nome della classe `A`.

Una direttiva di importazione che utilizzi il carattere * come, ad esempio:

```
import prog.io.*
```

è chiamata invece *importazione su richiesta*; essa aggiunge allo spazio dei nomi, i nomi di tutte le classi e di tutte le interfacce presenti nel package.

L'indicazione dei nomi completi delle classi consente di utilizzare nel codice classi diverse con lo stesso nome. Se, ad esempio, definiamo un altro package di nome `pack2`, contenente una classe di nome `A` di questo tipo:

```
package pack2;

public class A {
    public String toString() {
        return "classe pack2.A";
    }
}
```

possiamo utilizzare ambedue le classi all'interno dello stesso codice, come nella seguente applicazione:

```
import prog.io.*;

class Prova {
    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        pack1.A a1 = new pack1.A();
        out.println(a1.toString());

        pack2.A a2 = new pack2.A();
        out.println(a2.toString());
    }
}
```

Che cosa accade se due package importati su richiesta contengono classi o interfacce con lo stesso nome? Se ad esempio scriviamo le seguenti importazioni:

```
import pack1.*;
import pack2*;
```

nel codice successivo il compilatore non sarà più in grado di identificare a quale classe ci riferiamo quando utilizziamo il nome A; si ha quindi un conflitto di nomi. Il meccanismo dell'importazione su richiesta garantisce tuttavia che non ci saranno problemi, salvo che il nome della classe A compaia effettivamente nel codice: il compilatore carica il codice di una classe solo nel momento in cui essa viene effettivamente impiegata. Quando ciò accade il compilatore cerca il codice della classe nella directory corrente, nella directory che contiene le classi del package `java.lang` e quindi nelle directory specificate nelle importazioni su richiesta. Solo a questo punto si può verificare un conflitto di nomi.

Consideriamo ad esempio la seguente applicazione:

```
import prog.io.*;
import pack1.*;
import pack2*;
```

```

class Prova {
    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        A a1 = new A();
        out.println(a1.toString());
    }
}

```

In questo caso il compilatore segnala il conflitto di nomi così:

```

Prova.java:9: reference to A is ambiguous, both class pack2.A
in pack2 and class pack1.A in pack1 match
    A a1 = new A();
    ^

```

```

Prova.java:9: reference to A is ambiguous, both class pack2.A
in pack2 and class pack1.A in pack1 match
    A a1 = new A();
    ^

```

2 errors

Al contrario, questa applicazione è compilata correttamente:

```

import prog.io.*;
import pack1.*;

class Prova {
    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        A a1 = new A();
        out.println(a1.toString());

        pack2.A a2 = new pack2.A();
        out.println(a2.toString());
    }
}

```

In questo caso infatti la classe A del package pack1 è importata, mentre la classe A di pack2 è utilizzata tramite il suo nome completo.

Relativamente all'importazione delle classi osserviamo che il compilatore importa automaticamente le classi del package `java.lang` e le classi che si trovano nella directory in cui si esegue la compilazione; queste, come vedremo in seguito, appartengono a un package definito in modo automatico.

7.10 I modificatori di visibilità `public` e “amichevole”

Uno degli obiettivi dei package è quello di fornire un ambiente di protezione in cui sviluppare codice nascondendone i dettagli all'esterno. Tale meccanismo di protezione è realizzato tramite i modificatori di visibilità.

Abbiamo già detto che il modificatore di visibilità `private`, applicato ai membri e ai costruttori di una classe, ne limita la visibilità all'interno del codice della classe stessa, e fornisce il livello di protezione più forte.

Al contrario, il modificatore di visibilità `public` definisce il livello di protezione più debole. Le risorse dichiarate `public` (classi, interfacce, metodi, costruttori, campi) all'interno di un package sono accessibili a chiunque, e pertanto sono utilizzabili all'interno come all'esterno del package.

Esiste un terzo livello di visibilità la cui caratteristica può essere apprezzata solo nel momento in cui si definisce un package. Questo livello di visibilità è detto *amichevole* (o visibilità di package) ed è implicitamente applicato quando non viene indicato alcun modificatore. Consideriamo ad esempio il package `pack3`. Questo package contiene la classe `Visibile` così definita:

```
package pack3;

public class Visibile {
    public int x;
    int y;

    public Visibile() {
        this(1,2);
    }

    Visibile (int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String visibleFuori() {
        return "pack3.Visibile: metodo visibleFuori";
    }

    String nascostoFuori() {
        return "classe pack3.Visibile: metodo nascostoFuori";
    }
}
```

Si osservi che la classe è dichiarata `public`. Questo significa che la classe è visibile al di fuori del package. Analogamente il campo `x`, il costruttore privo di argomenti e il metodo

`visibleFuori`, che sono dichiarati `public`, sono visibili e quindi utilizzabili al di fuori del package.

Il campo `y`, il costruttore con due argomenti e il metodo `nascostoFuori` hanno invece visibilità amichevole, e quindi non sono visibili al di fuori del package. Per esempio, compilando la seguente applicazione:

```
import prog.io.*;
import pack3.*;

class Prova {
    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        Visibile v = new Visibile(3,3);
        out.println("y = " + v.y);
        out.println(v.nascostoFuori());
    }
}
```

il compilatore segnala i seguenti errori:

```
Prova.java:8: Visibile(int,int) is not public in pack3.Visibile;
cannot be accessed from outside package
```

```
    Visibile v = new Visibile(3,3);
```

```
Prova.java:9: y is not public in pack3.Visibile;
cannot be accessed from outside package
```

```
    out.println("y = " + v.y);
```

```
Prova.java:10: nascostoFuori() is not public in pack3.Visibile;
cannot be accessed from outside package
```

```
    out.println(v.nascostoFuori());
```

3 errors

Si osservi che il compilatore indica precisamente che i membri cui si fa riferimento nel codice sono definiti, ma non accessibili.

Lo stesso discorso vale per le classi (e le interfacce) con visibilità amichevole. Consideriamo la classe `Invisibile` del package `pack3` così definita:

```
package pack3;

class Invisibile {
    public String nascostoFuori() {
```

```
    return "pack3.Invisibile: metodo nascostoFuori";
}
}
```

Avendo visibilità amichevole, anch'essa non è visibile, e quindi non è utilizzabile all'esterno del package. Di conseguenza, compilando l'applicazione:

```
import prog.io.*;
import pack3.*;

class Prova {
    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        Invisibile v = new Invisibile();
        out.println(v.nascostoFuori());
    }
}
```

il compilatore segnala i seguenti errori:

```
Prova.java:8: pack3.Invisibile is not public in pack3;
cannot be accessed from outside package
    Invisibile v = new Invisibile();
    ^
Prova.java:8: pack3.Invisibile is not public in pack3;
cannot be accessed from outside package
    Invisibile v = new Invisibile();
    ^
Prova.java:8: Invisibile() is not public in pack3.Invisibile;
cannot be accessed from outside package
    Invisibile v = new Invisibile();
    ^
Prova.java:9: nascostoFuori() in pack3.Invisibile is not defined
in a public class or interface; cannot be accessed from outside package
    out.println(v.nascostoFuori());
    ^
4 errors
```

Si osservi che, sebbene il metodo `nascostoFuori` della classe `Invisibile` sia definito `public`, dato che è definito in una classe con visibilità amichevole, non risulta visibile al di fuori del package.

Tutto ciò che ha visibilità amichevole è invece visibile all'interno del package. Consideriamo, ad esempio, la seguente classe:

```

package pack3;

import prog.io.*;

public class ProvaDentro {
    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        Visibile v = new Visibile(3,3);
        out.println("y = " + v.y);
        out.println(v.nascostoFuori());

        Invisibile i = new Invisibile();
        out.println(i.nascostoFuori());
    }
}

```

Dato che fa parte del package pack3, la classe può utilizzare le risorse con visibilità amichevole del package, e quindi viene compilata.

Le classi di un package devono essere compilate insieme; ad esempio, la classe ProvaDentro dev'essere compilata con le classi Visibile e Invisibile che utilizza. Ciò può avvenire in due modi: o fornendo tutte le classi al compilatore con il comando

> javac Invisibile.java Visibile.java ProvaDentro.java

oppure utilizzando, più semplicemente, il comando

> javac *.java

La classe ProvaDentro definisce solo un metodo main; tuttavia richiedendone l'esecuzione con il comando

> java ProvaDentro

si osserva il seguente messaggio di errore:

```

Exception in thread "main" java.lang.NoClassDefFoundError:
ProvaDentro (wrong name: pack3/ProvaDentro)

```

La ragione di questo comportamento, come segnalato dalla Java Virtual Machine, è che il nome completo della classe contenuta nel file ProvaDentro.class è pack3.ProvaDentro, e non ProvaDentro. Per eseguire l'applicazione dovremo quindi utilizzare il nome completo della classe con il comando

> java pack3.ProvaDentro

In questo caso la Java Virtual Machine cerca il codice della classe da eseguire in un file di nome pack3/ProvaDentro.class (pack3\ProvaDentro.class nei sistemi DOS e Windows) a partire dai percorsi specificati nella variabile di sistema CLASSPATH.

Il package di default

Nei capitoli precedenti abbiamo sviluppato applicazioni composte da più classi senza includerle in un package. Java offre questa possibilità per agevolare la fase di sviluppo. In realtà, quando compiliamo delle classi che non sono esplicitamente incluse in un package, Java le tratta come se appartenessero a un package di default, senza nome, che include tutte le classi e tutte le interfacce definite nella medesima directory. Ne consegue che all'interno delle unità di compilazione presenti nella stessa directory è possibile utilizzare tutte le classi, le interfacce e i membri con visibilità amichevole definiti nelle unità di compilazione presenti nella directory.

7.11 Documentazione dei package

A ogni package può essere associata la documentazione che lo descrive integralmente. Questa documentazione dev'essere inserita in un file di nome `package.html` posto nella directory che contiene le unità di compilazione, cioè i file con estensione `.java` che costituiscono il package e che vengono utilizzati da `javadoc` per generare la documentazione.

Il contenuto del file `package.html` è interpretato da `javadoc` come un unico commento, scritto in HTML, che può contenere comandi di `javadoc`. Ci sono alcune limitazioni e alcune convenzioni relative al formato di questo file. Ad esempio la documentazione non deve contenere i delimitatori di documentazione `/**` e `*/` o linee che iniziano con asterischi. La prima frase nel corpo del documento HTML dovrebbe comprendere una descrizione succinta del package senza titolo o altro testo fra l'inizio del corpo del documento HTML, segnalato dal tag `<body>`, e la prima frase. Per i dettagli sulla struttura del file `package.html` rimandiamo al manuale di specifica di `javadoc`. Qui ci limitiamo a riportare, a titolo esemplificativo, il file `package.html` del package `prog.io` distribuito con questo testo.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Package prog.io</title>
  </head>

  <body>
    <p>Fornisce classi per la gestione dell'input/output da tastiera,  

       video, file. L'uso dei metodi forniti dalle classi di questo  

       package non richiede la conoscenza degli stream e delle  

       eccezioni.</p>

    @author Mauro Ferrari
    @author Giovanni Pighizzini
    @version 3.0
  </body>
</html>
```

Per costruire la documentazione relativa a un intero package è possibile utilizzare il comando javadoc fornendogli come argomento il nome del package. La documentazione del package `prog.io` può essere generata con il comando

```
javadoc prog.io
```

In conclusione di questo paragrafo presentiamo due marcatori di javadoc che vengono utilizzati per creare collegamenti fra varie parti della documentazione.

- **@see riferimento**

Viene utilizzato per inserire un collegamento ipertestuale a un'altra parte della documentazione generata da javadoc. I collegamenti corrispondenti ai marcatori `@see` vengono inseriti in un'apposita sezione, “*Vedere anche*” (“*See also*”), alla fine della documentazione della classe corrispondente. *riferimento* può avere una delle forme seguenti.

- *nome_package*

Per inserire un collegamento al package specificato.

- *nome_package.nome_classe*

Per inserire un collegamento alla classe specificata. Se *nome_package* non è presente, javadoc lo interpreta come un collegamento alla classe *nome_classe* all'interno dello stesso package in cui compare la classe dov'è utilizzato il marcitore.

- *nome_package.nome_classe#nome_membro*

Per inserire un collegamento al membro specificato dopo il carattere # della classe specificata. Se *nome_package* non è presente, allora javadoc lo interpreta come un collegamento al membro della classe *nome_classe* all'interno dello stesso package dove compare la classe in cui è inserito il commento di documentazione. Se *nome_package.nome_classe* non è presente, allora javadoc lo interpreta come un collegamento al membro della stessa classe in cui è inserito il marcitore.

- **{@link riferimento}**

Analogo a `@see`, ma il riferimento viene inserito direttamente nel testo (all'interno di una coppia di “parentesi graffe”). *riferimento* va definito con le stesse modalità descritte per `@see`.

7.12 UML: membri di una classe

Per concludere questo capitolo presentiamo in dettaglio la notazione di UML inerente alla rappresentazione dei membri di una classe. Nel Capitolo 6 abbiamo visto che una classe viene rappresentata graficamente mediante un rettangolo diviso in tre compatti in cui sono indicati, rispettivamente dall'alto in basso, il nome della classe, i campi (*attributi* nella terminologia di UML), i costruttori e i metodi (*operazioni* nella terminologia di UML). La sintassi dei campi di UML è la seguente:

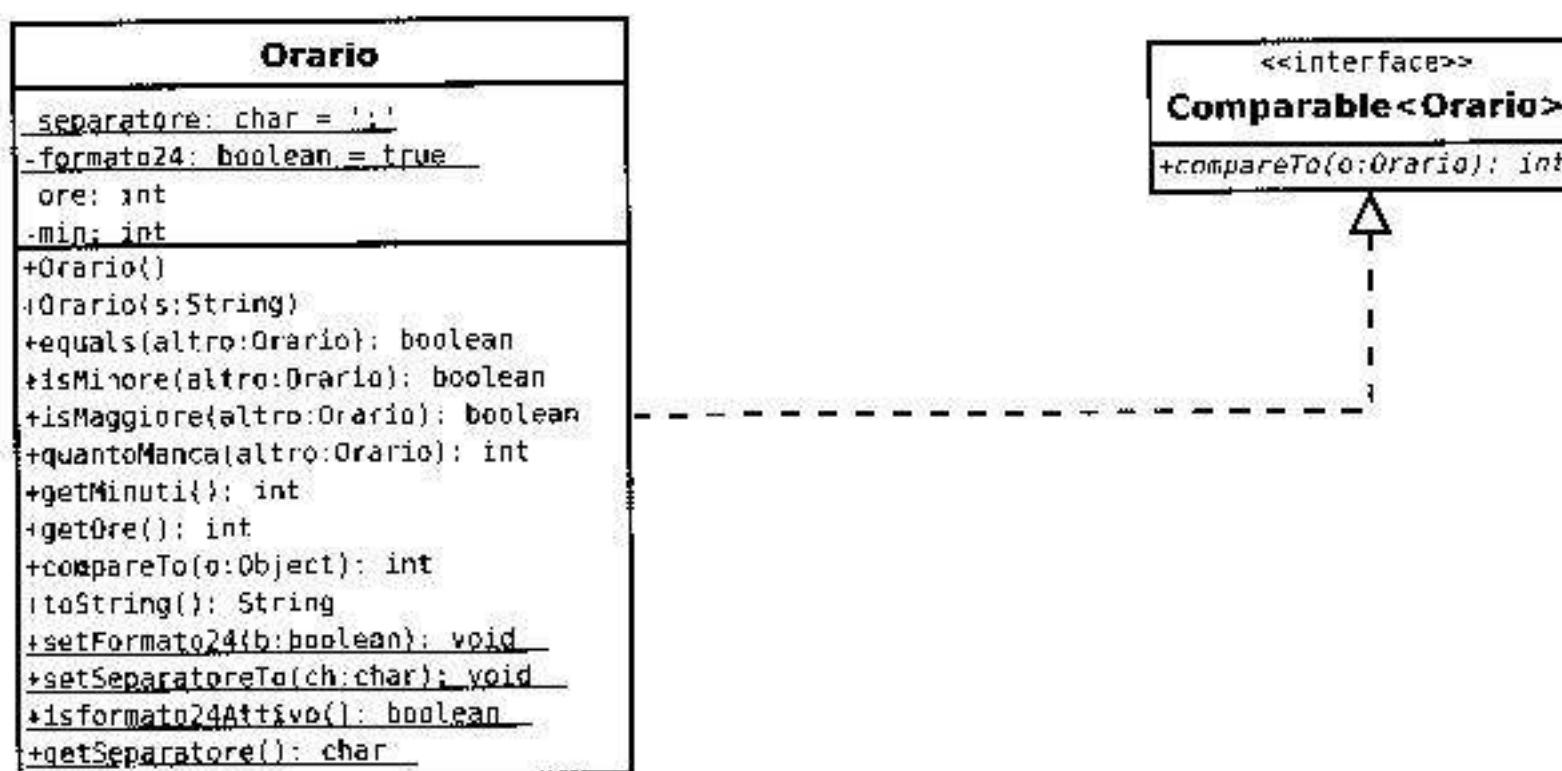


Figura 7.3 La classe Orario.

visibilità nome_campo : tipo = valore

visibilità indica la visibilità del campo (attributo). Come in Java, se non è indicata, è *amichevole*, cioè il campo è visibile, e quindi utilizzabile da tutte le classi che appartengono al medesimo package. Gli altri valori possibili sono:

- +: corrisponde alla visibilità definita dal modificatore `public` di Java;
- -: corrisponde alla visibilità definita dal modificatore `private` di Java;
- #: corrisponde alla visibilità definita dal modificatore `protected` di Java, che studieremo nel prossimo capitolo.

valore indica invece il valore con cui viene inizializzato il campo all'atto della creazione di un'istanza della classe. L'inizializzazione è opzionale.

Per quel che riguarda le operazioni, la sintassi completa è la seguente:

visibilità nome_operazione (lista_dei_parametri) : tipo_restituito

dove *visibilità* è definibile come nel caso dei campi.

In UML i membri statici sono indicati con la stessa sintassi dei membri non statici, ma vengono sottolineati.

Nella Figura 7.3 forniamo a titolo d'esempio la rappresentazione completa della classe `Orario` che abbiamo codificato nel Paragrafo 7.4, con l'aggiunta dell'implementazione dell'interfaccia `Comparable` proposta come esercizio.

Esercizi

7.15 Inserite l'applicazione Agenda sviluppata nell'Esercizio 7.12 in un package e scrivete la documentazione completa del package.

Estensione delle classi

Nel capitolo precedente abbiamo mostrato come implementare le classi partendo da zero: per ogni classe abbiamo definito in modo esplicito tutti i suoi metodi e i suoi campi, caratterizzando lo stato e il comportamento degli oggetti. Di frequente è necessario realizzare oggetti il cui stato e comportamento siano molto simili a quelli già implementati da qualche classe, o ne siano semplicemente casi particolari. Il meccanismo di estensione delle classi del linguaggio Java fornisce uno strumento per definire nuove classi a partire da quelle esistenti, in cui lo stato e il comportamento definiti dalla classe esistente (superclasse) vengono *ereditati* dalla nuova classe (sottoclasse).

8.1 Ereditarietà e implementazione di sottoclassi

Gli esempi che svilupperemo in questo capitolo riguardano principalmente l'implementazione di alcune classi presentate nel Capitolo 6, dove abbiamo introdotto la gerarchia delle classi.

Il primo esempio che illustriamo è l'implementazione della classe `Quadrato` come estensione della classe `Rettangolo`, i cui metodi e costruttori sono stati elencati nel Paragrafo 6.1. Il meccanismo di estensione delle classi consente di definire la classe `Quadrato` recuperando quanto possibile dell'implementazione fornita nella classe `Rettangolo`. In particolare implementeremo `Quadrato` senza conoscere nulla dell'implementazione di `Rettangolo`, ma basandoci solo sul suo contratto. In generale, è possibile estendere classi senza conoscerne l'implementazione, ma solo il comportamento.

Prima di tutto specifichiamo nell'intestazione della classe `Quadrato` che essa *estende* `Rettangolo` come segue:

```
class Quadrato extends Rettangolo {  
    ...  
}
```

In questo contesto, la parola chiave `extends` (*estende*) indica che la classe `Quadrato` è ottenuta dalla classe `Rettangolo` estendendone stato e comportamento, cioè aggiungendo campi o metodi. Dunque gli oggetti della classe `Quadrato` sono oggetti della classe `Rettangolo` che

“sanno fare qualcosa in più”. Ogni **Quadrato** è anche un **Rettangolo**, ma non è vero il contrario. La classe **Quadrato** è in questo senso una sottoclasse della classe **Rettangolo**. A sua volta **Rettangolo** è una *superclasse* di **Quadrato**.

La classe **Quadrato**, per il solo fatto di essere definita come estensione della classe **Rettangolo**, dispone già di tutti i campi e di tutti i metodi definiti nella classe **Rettangolo**, cioè *eredita* i membri della classe **Rettangolo** (come vedremo più avanti, ciò non implica però che questi membri siano accessibili nel codice di **Quadrato**). Al contrario dei campi e dei metodi, i costruttori non vengono ereditati.

Procedendo nell’implementazione della classe **Quadrato**, ne definiamo per prima cosa il costruttore che, in questo caso, deve ricevere un parametro **x** di tipo **double** e costruire un quadrato il cui lato sia uguale a **x**.

In generale la costruzione di un oggetto di una sottoclasse avviene costruendo un oggetto della superclasse, il quale è poi adattato alle esigenze della sottoclasse. In questo caso è sufficiente costruire un **Rettangolo** con base e altezza uguali al valore del parametro **x**. A tale scopo bisogna che il costruttore di **Quadrato** invochi quello di **Rettangolo** fornendo, per ambedue gli argomenti, il valore di **x**.

Per invocare il costruttore della superclasse, Java prevede la parola riservata **super**. Come **this**, descritta nel capitolo precedente, anche **super** assume due significati diversi a seconda del contesto in cui compare. Nel contesto del costruttore di una sottoclasse, **super** fa riferimento al costruttore della superclasse. Pertanto il costruttore della classe **Quadrato** sarà:

```
public Quadrato(double x) {
    super(x, x);
}
```

La classe **Quadrato**, con il costruttore definito qui, è compilabile e utilizzabile come una qualunque altra classe. Tutti i metodi della classe **Rettangolo** sono invocabili su oggetti della classe **Quadrato**.

In base alla specifica presentata nel Capitolo 6, la classe **Quadrato** deve possedere il metodo **getLato**, assente in **Rettangolo**. Il testo di questo metodo sarà dunque scritto nel corpo della classe **Quadrato**.

Vediamo come implementare il metodo. Esso deve restituire la lunghezza del lato del quadrato che lo esegue. Poiché un quadrato è un rettangolo con la base uguale all’altezza, è sufficiente che il metodo restituisca, indifferentemente, il valore della base o quello dell’altezza del rettangolo. Come reperire tale valore? Anche se non conosciamo il modo in cui i rettangoli sono rappresentati all’interno della classe **Rettangolo**, sappiamo che questa dispone dei metodi **getBase** e **getAltezza**. Grazie al meccanismo dell’ereditarietà essi sono disponibili anche per **Quadrato**. Per fornire il valore del proprio lato, un oggetto di tipo **Quadrato** può quindi eseguire il proprio metodo **getBase** o **getAltezza** e restituire il risultato ottenuto in questo modo:

```
public double getLato() {
    return this.getBase();
}
```

o, più semplicemente, come

```
public double getLato() {
    return getBase();
}
```

Ecco il codice completo di questa prima versione della classe Quadrato:

```
public class Quadrato extends Rettangolo {
    //COSTRUTTORI
    public Quadrato(double x) {
        super(x, x);
    }

    //METODI
    public double getLato() {
        return getBase();
    }
}
```

Sottolineiamo nuovamente che la classe Quadrato è definita e implementata sapendo solo *che cosa fa* la classe Rettangolo, e non *come* è fatta. In generale è possibile (e opportuno) implementare una sottoclasse a prescindere dall'implementazione della superclasse. In questo modo eventuali modifiche all'implementazione della superclasse non comprometteranno l'implementazione della sottoclasse.

Ecco un esempio di esecuzione dell'applicazione ProvaQuadrato del Paragrafo 6.3 con questa versione di Quadrato:

```
Quanto è lungo il lato del quadrato? 6
Quadrato letto: base = 6.0, altezza = 6.0
L'area è 36.0
Il perimetro è 24.0
```

Si noti che il messaggio riportato nella seconda riga fornisce sia il valore della base sia quello dell'altezza, mentre sarebbe opportuno fornire unicamente la lunghezza del lato. Ciò dipende dal fatto che in questa prima versione di Quadrato non abbiamo ridefinito il metodo `toString`. Quindi esso viene ereditato da Rettangolo.

Scriviamo ora una nuova versione di Quadrato in cui il metodo `toString` sia ridefinito per fornire la sola indicazione del lato. Il metodo richiama a sua volta il metodo `getLato`:

```
public class Quadrato extends Rettangolo {
    //COSTRUTTORI
    public Quadrato(double x) {
        super(x, x);
    }
```

```
//METODI
public double getLato() {
    return getBase();
}

public String toString() {
    return "lato = " + getLato();
}
}
```

Ecco un esempio di esecuzione di ProvaQuadrato con questa nuova versione della classe Quadrato:

```
Quanto è lungo il lato del quadrato? 6
Quadrato letto: lato = 6.0
L'area è 36.0
Il perimetro è 24.0
```

I metodi eseguibili da oggetti della classe Quadrato possono essere suddivisi in tre gruppi:

- metodi *definiti* per la prima volta nella classe Quadrato, come il metodo `getLato`;
- metodi *ridefiniti* nella classe Quadrato, come il metodo `toString`: hanno la segnatura uguale a quella di un metodo della superclasse, ma lo ridefiniscono;
- metodi *ereditati* dalla superclasse che non siano ridefiniti nella sottoclasse, come i metodi `getArea` e `getPerimetro`.

8.2 Costruttori e gerarchia delle classi

Il costruttore che abbiamo scritto per la classe Quadrato invoca, mediante il riferimento `super`, quello della superclasse Rettangolo. D'altra parte, nell'implementazione della classe Frazione presentata nel capitolo precedente abbiamo fornito due costruttori: l'implementazione di uno di essi è basata su un'invocazione dell'altro tramite l'utilizzo del riferimento `this`.

Generalmente un costruttore non opera mai da solo, ma si avvale sempre dell'ausilio di un altro costruttore che può essere della stessa classe o della superclasse. Più precisamente, per scrivere e utilizzare correttamente i costruttori è necessario tenere conto delle seguenti regole.

- (1) Se in una classe non si definiscono costruttori, viene automaticamente aggiunto un costruttore privo di argomenti (e il cui corpo è vuoto). Dunque *ogni classe possiede almeno un costruttore*. Se, al contrario, nella classe viene definito esplicitamente almeno un costruttore, il costruttore privo di argomenti *non* viene aggiunto automaticamente.

- (2) Se un costruttore non invoca esplicitamente un costruttore della sua superclasse (utilizzando `super`) o un costruttore della stessa classe (utilizzando `this`), all'inizio del costruttore viene automaticamente invocato il costruttore privo di argomenti della superclasse.¹
- (3) In un costruttore, l'invocazione di un costruttore della superclasse o di un altro costruttore della stessa classe può trovarsi solo come prima istruzione.

Illustriamo le regole precedenti mediante alcuni esempi. Supponiamo di disporre di una classe `A` nella cui implementazione non sia stato scritto esplicitamente alcun costruttore. In base alla prima regola, il compilatore aggiungerà automaticamente il seguente costruttore:

```
public A() {  
}
```

La seconda regola stabilisce che, senza l'invocazione esplicita di un costruttore della superclasse mediante `super`, o di un altro costruttore della stessa classe mediante `this`, all'inizio di un costruttore viene aggiunta implicitamente la linea:

```
super();
```

Quindi, il costruttore di `A` indicato sopra invocherà il costruttore privo di argomenti della superclasse, cioè sarà equivalente a:

```
public A() {  
    super();  
}
```

Se `A` è, ad esempio, una sottoclasse diretta di `Object`, il costruttore invocato dalla chiamata `super()` sarà quello privo di argomenti di `Object`.

Se invece la superclasse non disponesse di un costruttore privo di argomenti, il compilatore segnalerebbe un errore.

Consideriamo un secondo esempio. La classe `Quadrato` dispone di un proprio costruttore. Pertanto, in base alla prima regola, il costruttore privo di argomenti *non viene aggiunto* automaticamente. Supponiamo di volerlo definire in modo che costruisca un quadrato di lato nullo. Scrivendo il costruttore come:

```
public Quadrato() {  
}
```

il compilatore segnalerà un errore.

Infatti, in base alla seconda regola data, non essendovi un'invocazione esplicita dell'altro costruttore di `Quadrato` o di quello di `Rettangolo`, il compilatore aggiungerà automaticamente un'invocazione del costruttore privo di argomenti della classe `Rettangolo`.

¹ C'è una sola eccezione a questa regola: il costruttore della classe `Object` predisponde l'oggetto senza alcun ausilio.

In altre parole, il compilatore tratterà il costruttore come se fosse:

```
public Quadrato() {
    super();
}
```

La classe `Rettangolo` non dispone però di un costruttore privo di argomenti. Di conseguenza la chiamata `super()` non è compilabile.

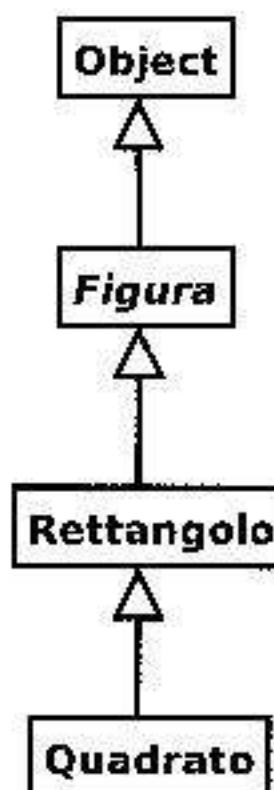
Ricorrendo all'altro costruttore della classe, il nuovo costruttore di `Quadrato` avrebbe potuto essere scritto come:

```
public Quadrato() {
    this(0);
}
```

o in alternativa, ricorrendo al costruttore della superclasse, come:

```
public Quadrato() {
    super(0, 0);
}
```

Il senso delle regole precedenti può essere compreso meglio rifacendosi alla gerarchia delle classi. Per esemplificare il discorso facciamo riferimento al ramo della gerarchia contenente, dall'alto verso il basso, le classi `Object`, `Figura`, `Rettangolo` e `Quadrato`.



Ognuna di queste classi è un'estensione della precedente: si può costruire un'istanza della sottoclasse a partire da un'istanza della superclasse, eventualmente modificandola. Ad esempio un quadrato può essere ottenuto costruendo un rettangolo; un rettangolo può essere costruito a partire da una figura generica con una base e un'altezza; una figura generica può essere costruita da un oggetto generico.

In sostanza, per costruire un'istanza di `Quadrato` intervengono via via i costruttori delle superclassi fino a risalire al costruttore più generico, cioè a quello di `Object`, che predisponde inizialmente l'oggetto. Quest'oggetto viene poi "modellato" dai costruttori delle sottoclassi fino alla costruzione definitiva dell'istanza di `Quadrato`.

In pratica ogni costruttore delega parte del proprio compito a un costruttore della superclasse (o talvolta della classe stessa, se si usa il riferimento `this`). Una volta che il costruttore della superclasse ha prodotto l'oggetto, il costruttore della sottoclasse può modificarlo, aggiungendovi ad esempio ulteriori campi. In ogni caso un costruttore imprime sull'oggetto il proprio *marchio di fabbrica*, cioè il nome della classe cui appartiene. Questa informazione è indispensabile alla Java Virtual Machine durante l'esecuzione per conoscere il tipo effettivo dell'oggetto, ad esempio per calcolare il valore dell'operatore `instanceof` o per selezionare il metodo da eseguire in caso di polimorfismo.

Concludiamo questo paragrafo con alcune osservazioni relative alle classi astratte. Ricordiamo che esse non possono essere istanziate. Tuttavia ogni classe astratta (come del resto tutte le altre classi) possiede sempre almeno un costruttore. Le regole indicate all'inizio del paragrafo valgono anche per queste: se la classe astratta non contiene esplicitamente un costruttore, verrà aggiunto quello privo di argomenti.

A che cosa servono i costruttori delle classi astratte se esse non possono essere istanziate? Si può rispondere facilmente riconsiderando quanto detto in precedenza: per costruire un oggetto di una sottoclasse viene *sempre* richiamato (implicitamente o esplicitamente) il costruttore della superclasse. Quindi i costruttori delle classi astratte vengono utilizzati nel processo di fabbricazione degli oggetti delle sottoclassi concrete.

8.3 Il riferimento super

Nel capitolo precedente abbiamo utilizzato il riferimento `this` in due modi:

- per richiamare, all'inizio di un costruttore, un altro costruttore della stessa classe;
- per riferirsi all'oggetto in esecuzione al fine di accedere ai suoi campi (come nella scrittura `this.num` utilizzata nella classe `Frazione`), o per richiamare da un suo metodo altri metodi (come nella chiamata del metodo `meno` dal metodo `isMinore`).

Anche per il riferimento `super` esistono usi analoghi: il primo, inherente al richiamo del costruttore della superclasse nel codice del costruttore della sottoclasse, è stato illustrato nel primo paragrafo di questo capitolo; il secondo, in particolare per quel che riguarda l'invocazione dei metodi, lo tratteremo ora.

Consideriamo il metodo `getLato` della classe `Quadrato`:

```
public double getLato() {
    return getBase();
}
```

In mancanza di un riferimento esplicito è sempre sottinteso il riferimento `this`. Pertanto la semplice invocazione `getBase()` è sintatticamente equivalente a `this.getBase()`, il cui significato è il messaggio che l'oggetto in esecuzione invia a se stesso: “esegui il tuo metodo `getBase`”.

Il significato dell'invocazione `super.getBase()` è invece il messaggio (invia sempre dall'oggetto in esecuzione a se stesso): “esegui il metodo `getBase` della superclasse”. In questo caso il metodo eseguito è lo stesso di prima. Infatti, poiché non è stato ridefinito nella classe `Quadrato`, il metodo `getBase` di un oggetto è proprio quello ereditato dalla superclasse `Rettangolo`.

Se invece un metodo viene ridefinito nella sottoclasse, è possibile riscontrare differenze. Supponiamo infatti di scrivere nella classe `Quadrato` i seguenti due metodi, che restituiscono una stringa che descrive l'oggetto:

```
public String getDescrizione1() {  
    return this.toString();  
}  
  
public String getDescrizione2() {  
    return super.toString();  
}
```

Il primo metodo (in cui avremmo potuto lasciare sottinteso il riferimento `this`) si limita a richiamare il metodo `toString` dell'oggetto in esecuzione e a riportarne il risultato. Pertanto, se l'oggetto è un'istanza di `Quadrato`, il metodo restituirà una stringa del tipo:

```
lato = 6.0
```

Il secondo metodo, invece, richiama mediante il riferimento `super` il metodo `toString` corrispondente alla superclasse, cioè a `Rettangolo`. Pertanto in questo caso l'esecuzione restituirà una stringa del tipo:

```
base = 6.0, altezza = 6.0
```

In pratica, mediante il riferimento `super`, la ricerca del metodo da eseguire non parte dalla classe effettiva dell'oggetto, ma dalla superclasse della classe in cui è scritta l'invocazione stessa.

Il riferimento `super` può anche essere utilizzato davanti ai nomi di variabili per l'accesso ai campi corrispondenti alla superclasse (purché non siano resi inaccessibili, ad esempio tramite il modificatore `private`). Analizzeremo questo aspetto più avanti nel capitolo.

8.4 Ereditarietà e stato degli oggetti

Consideriamo nuovamente la classe `Rettangolo` e la sua sottoclasse `Quadrato`. Nel Paragrafo 6.4 abbiamo scritto che `Quadrato` eredita i campi definiti nella classe `Rettangolo`. Questo

non implica che tali campi siano direttamente accessibili.² In realtà, siamo stati in grado di implementare Quadrato senza fare alcun riferimento ai campi di Rettangolo, basandoci invece esclusivamente sulla relazione geometrica che intercorre tra quadrati e rettangoli: un quadrato è un rettangolo con base e altezza uguali. Com'è fatto un oggetto della classe Quadrato? Come è definito il suo stato? Poiché la classe Quadrato estende Rettangolo (e nella classe Quadrato non sono definiti ulteriori campi), lo stato di un oggetto della classe Quadrato è definito da un oggetto della classe Rettangolo. In altre parole, un oggetto della classe Quadrato contiene al suo interno un Rettangolo.

Consideriamo ora la seguente sottoclass SottoRettangolo di Rettangolo:

```
public class SottoRettangolo extends Rettangolo {
    private int x;
    private String w;

    ...costruttori e metodi...
}
```

A differenza della classe Quadrato, nella quale non era definito alcun campo, nella classe SottoRettangolo sono definiti i due campi x e w. Pertanto, un oggetto di questa classe contiene al suo interno un oggetto di tipo Rettangolo (in quanto SottoRettangolo estende Rettangolo) più i due campi x e w. L'oggetto di tipo Rettangolo, cioè la parte "ereditata", è accessibile tramite il riferimento super. In particolare se ne possono invocare i metodi (cioè i metodi previsti dalla classe Rettangolo, anche se sono stati ridefiniti, come evidenziato nel paragrafo precedente). I campi x e w sono invece direttamente accessibili all'interno del codice della classe SottoRettangolo.

Il procedimento con cui lavorano i costruttori, descritto nel Paragrafo 8.2, rispecchia proprio questo legame tra lo stato di un oggetto della sottoclass e quello della superclasse. Ad esempio, per costruire un oggetto della classe SottoRettangolo è necessario costruire un oggetto della classe Rettangolo a cui vengono aggiunti i due campi x e w.

Esempio

Consideriamo la seguente sottoclass Beta, che estende la classe Data del package prog.utili:

```
import prog.utili.Data;

public class Beta extends Data {
    private int w;
    private static int k = 4;

    public Beta(int x) {
```

² Come mostreremo più avanti implementando la classe Rettangolo, in questo caso non lo sono in quanto dichiarati privati.

```

super(x + 2, 5, 2004);
w = x;
k++;
}

public Beta(String s) {
    w = s.length();
    k++;
}

public int getGiorno() {
    return super.getGiorno() + w;
}

public static int getStatico() {
    return k;
}
}

```

Nella classe Beta sono definiti il campo w e il campo statico k. Ricordiamo che il campo statico appartiene all'intera classe e, dunque, non fa parte dello stato degli oggetti. Poiché Beta estende Data, un'istanza della classe Beta è formato da un'istanza di Data più un campo w di tipo int. Possiamo schematizzare l'oggetto come segue:

Istanza di Beta:

Istanza di Data: ...una data...
w: ...un intero...

Osserviamo i costruttori della classe Beta. Il primo invoca esplicitamente un costruttore della superclasse. Il secondo, al contrario, non contiene alcuna invocazione di costruttori della superclasse o della classe stessa. Pertanto, il compilatore aggiungerà automaticamente al suo inizio un'invocazione super().

Vogliamo ora determinare ciò che viene visualizzato dalla seguente applicazione:

```

import prog.utili.Data;
import prog.io.*;

class Gamma {
    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        Data a = new Data(22, 6, 2004);
        out.println(a.getGiorno());
        a = new Beta("Topo");
    }
}

```

```

    out.println(a.getGiorno());
    Beta b = new Beta("topo");
    out.println(b.getGiorno());
    out.println(Beta.getStatico());
}
}

```

L'applicazione utilizza le classi Beta e Data. In particolare, la classe Beta contiene un campo statico `k` di tipo `int`, inizializzato a 4.³

Dopo la creazione del canale di comunicazione con il monitor, viene creato un oggetto di tipo Data, che rappresenta la data 22 giugno 2004. Il riferimento a tale oggetto viene memorizzato nella variabile `a`.⁴ Nell'istruzione successiva viene visualizzato il risultato del metodo `getGiorno`, invocato tramite il riferimento `a`. In base al contratto della classe Data, a cui appartiene l'oggetto riferito da `a`, il metodo restituisce 22.

Nella riga successiva viene costruito un oggetto di tipo Beta e il riferimento viene posto in `a`. L'oggetto viene costruito invocando il costruttore che riceve un parametro di tipo `String`. In particolare viene fornita la stringa "topo". Come evidenziato in precedenza, il costruttore invoca prima di tutto, mediante `super()`, il costruttore privo di argomenti della superclasse Data. In base al proprio contratto, tale costruttore fabbrica un oggetto che rappresenta la data del momento in cui viene invocato, cioè (per chi sta scrivendo) 9 aprile 2005. Al campo `w` viene assegnata la lunghezza della stringa parametro, cioè 4; il campo statico `k` della classe viene poi incrementato, diventando 5. Pertanto, lo stato dell'oggetto costruito, a cui farà riferimento `a`, sarà il seguente:

```

Istanza di Beta:
Istanza di Data: 9 aprile 2005
w: 4

```

Nell'istruzione successiva viene invocato il metodo `getGiorno`. L'oggetto al quale viene richiesta l'esecuzione del metodo è un'istanza di Beta. In questa classe il metodo è ridefinito. Il metodo `getGiorno` di Beta invoca il metodo della superclasse che, in questo caso, restituisce 9, e aggiunge il valore del campo `w`, in questo caso 4, restituendo il risultato al chiamante. Pertanto il risultato visualizzato è 13.⁵ Nell'istruzione successiva viene di nuovo costruita un'istanza di Beta, invocando il costruttore con parametro `int`. In questo caso, il riferimento all'oggetto costruito verrà posto nella variabile `b`. Simulando le operazioni effettuate dal costruttore, possiamo osservare che l'oggetto costruito ha il seguente stato:

```

Istanza di Beta:
Istanza di Data: 17 maggio 2004
w: 15

```

³ È bene che il lettore segua questa discussione disegnando, man mano, le variabili, i campi e gli oggetti utilizzati. Inizialmente vi sarà il campo statico `k`, associato all'intera classe Beta, contenente 4.

⁴ Il lettore dovrebbe a questo punto disegnare un riquadro e scriverci 22 giugno 2004. Inoltre dovrebbe disegnare una freccia, ovvero, il riferimento, dalla variabile `a` al riquadro.

⁵ Chiaramente il risultato dipende dalla data corrente.

Inoltre, il campo statico k della classe Beta viene nuovamente incrementato, raggiungendo il valore 6.

Simulando, in maniera analoga a quanto visto precedentemente, la successiva chiamata di `getGiorno`, possiamo concludere che la successiva istruzione visualizza il valore 32.

Infine, l'ultima istruzione visualizza il valore del campo statico k della classe Beta, cioè 6.

Esercizi

8.1 Considerate le seguenti classi:

```
public class Alfa {  
    private int x;  
    private static int y = 5;  
  
    public Alfa() {  
        y++;  
    }  
  
    public void inc() {  
        x++;  
    }  
  
    public int getTotale() {  
        return x + y;  
    }  
}  
  
public class Beta extends Alfa {  
    private int w = 7;  
    public void inc() {  
        super.inc();  
        w++;  
    }  
  
    public int getTotale() {  
        return super.getTotale() + w;  
    }  
}  
  
class Gamma {  
    public static void main(String[] args) {  
        ConsoleOutputManager out = new ConsoleOutputManager();
```

```

Alfa a = new Alfa();
a.inc();
out.println(a.getTotale());
a = new Beta();
a.inc();
out.println(a.getTotale());
Beta b = new Beta();
b.inc();
out.println(b.getTotale());
out.println(a.getTotale());
}
}

```

Indicate i risultati visualizzati sullo schermo dal metodo `main` della classe `Gamma` (ricordate di importare la classe `ConsoleOutputManager` prima di compilare la classe `Gamma`). Verificate poi le vostre risposte facendo eseguire il metodo `main` sulla vostra macchina.

- 8.2 Considerate le seguenti classi che utilizzano la classe `Importo` del package `prog.utili` (che dovrà essere opportunamente importata insieme alla classe `ConsoleOutputManager`):

```

public class Alfa extends Importo {
    private String w = "ratto";
    private static int k = 1;
    public Alfa() {
        super(10);
        k++;
    }

    public Alfa(String s) {
        this();
        w = s;
    }

    public int getIntero() {
        return this.getEuro() + w.length();
    }

    public static int getStatico() {
        return k;
    }
}

```

```

class Prova {
    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        println(Alfa.getStatico());
        Alfa a = new Alfa();
        out.println(a.getIntero());
        a = new Alfa("pappagallo");
        out.println(a.getIntero());
        out.println(Alfa.getStatico());
    }
}

```

Indicate i risultati visualizzati sullo schermo dal metodo `main` della classe `Prova`. Verificate poi le vostre risposte facendo eseguire il metodo `main` sulla vostra macchina.

8.5 Overloading dei metodi

Come abbiamo osservato utilizzando classi già implementate, in Java si possono definire più metodi con il medesimo nome, ma con segnature differenti (overloading).⁶

Si consideri ad esempio il seguente metodo per il calcolo del valore assoluto di un numero `double`:

```

public static double valoreAssoluto(double x) {
    if (x > 0)
        return x;
    else
        return -x;
}

```

Questo metodo, applicato a un valore `double`, restituisce un risultato `double`. Applicando il metodo a un valore `int`, il risultato sarà sempre `double`. Volendo disporre di un metodo per calcolare il valore assoluto di numeri `int` (con risultato di tipo `int`), possiamo utilizzare l'overloading definendo nella stessa classe:

```

public static int valoreAssoluto(int x) {
    if (x > 0)
        return x;
    else
        return -x;
}

```

⁶ L'analisi presentata in questo paragrafo è valida anche per l'overloading dei costruttori.

In base al tipo dell'argomento utilizzato nella chiamata, il compilatore riconosce quale metodo dovrà essere eseguito. Si osservi che il secondo metodo si potrebbe anche scrivere come:

```
public static int valoreAssoluto(int x) {
    return (int) valoreAssoluto((double) x);
}
```

Nell'istruzione `return` viene invocato il metodo `valoreAssoluto`; il cast nell'argomento di `x` a `double` fa in modo che venga chiamato il metodo `valoreAssoluto` definito per i `double`. Tale metodo restituisce un risultato `double` che è poi forzato a `int`.⁷

L'overloading viene risolto *in fase di compilazione*. Più precisamente, in fase di compilazione viene scelta *la segnatura* del metodo da eseguire sulla base del tipo del riferimento utilizzato per invocare il metodo (analizzando cioè i metodi disponibili per quel riferimento) e degli argomenti indicati nella chiamata. Per effettuare questa scelta, il compilatore utilizza un algoritmo che cerca, tra le differenti segnature disponibili, quella che più si avvicina alla chiamata. Ad esempio, per la chiamata

`r.m(2)`

dove `r` è un riferimento e `m` è il nome di un metodo, il compilatore cercherà quella adatta fra tutte le segnature di metodi di nome `m` disponibili per il tipo di `r`.

Supponiamo che siano disponibili una segnatura con parametro `long` e una con parametro `double`. Il compilatore scelgerà quella con parametro `long`: il tipo `int` utilizzato per l'argomento è infatti più vicino al tipo `long` che al tipo `double`. Nel prossimo paragrafo esamineremo in dettaglio il problema della scelta della segnatura del metodo da eseguire, che è direttamente collegata all'overloading, e il problema della scelta del metodo da eseguire, collegata invece all'overriding.

8.6 Overriding, overloading e scelta del metodo da eseguire

Ricordiamo che con l'*overriding* si riscrive in una sottoclasse un metodo della superclasse con *la stessa segnatura*, mentre con l'*overloading* è possibile definire metodi con lo stesso nome ma con *segnature differenti*. Abbiamo visto che il compilatore risolve l'overloading stabilendo quale metodo si debba eseguire sulla base degli argomenti utilizzati nella chiamata. In realtà, se per uno stesso metodo c'è sia overloading sia overriding, il compilatore può stabilire solo la *segnatura* del metodo da eseguire (*early binding*), mentre la decisione relativa al metodo effettivo, tra quelli con la segnatura selezionata, viene rimandata all'esecuzione (*late binding*). Trattiamo ora questi aspetti dettagliatamente, precisando quanto avviene in compilazione e quanto avviene in esecuzione.

⁷ È possibile anche che un metodo richiami se stesso. Questa eventualità, che analizzeremo in maniera approfondita successivamente, prende il nome di *ricorsione*.

Per l'analisi che segue utilizzeremo tre riferimenti: alfa, beta e gamma, dichiarati come:

```
A alfa;  
B beta;  
C gamma;
```

dove A, B e C sono queste classi:

```
public class A {  
    private long a;  
  
    public void assegna(long x) {  
        a = x;  
    }  
  
    public String toString() {  
        return "a = " + a;  
    }  
}  
  
public class B extends A {  
    private int b1;  
    private double b2;  
  
    public B() {  
        b1 = 1;  
        b2 = 1;  
    }  
  
    public void assegna(int x) {  
        b1 = x;  
    }  
  
    public void assegna(double x) {  
        b2 = x;  
    }  
  
    public String toString() {  
        return super.toString() + ", b1 = " + b1 + ", b2 = " + b2;  
    }  
}  
  
public class C extends B {  
    private int c1;
```

```
private double c2;

public void assegna(int x) {
    c1 = x;
}

public void assegna(double x) {
    c2 = x;
}

public String toString() {
    return super.toString() + ", c1 = " + c1 + ", c2 = " + c2;
}
```

La gerarchia di queste classi è riassunta nella Figura 8.1.

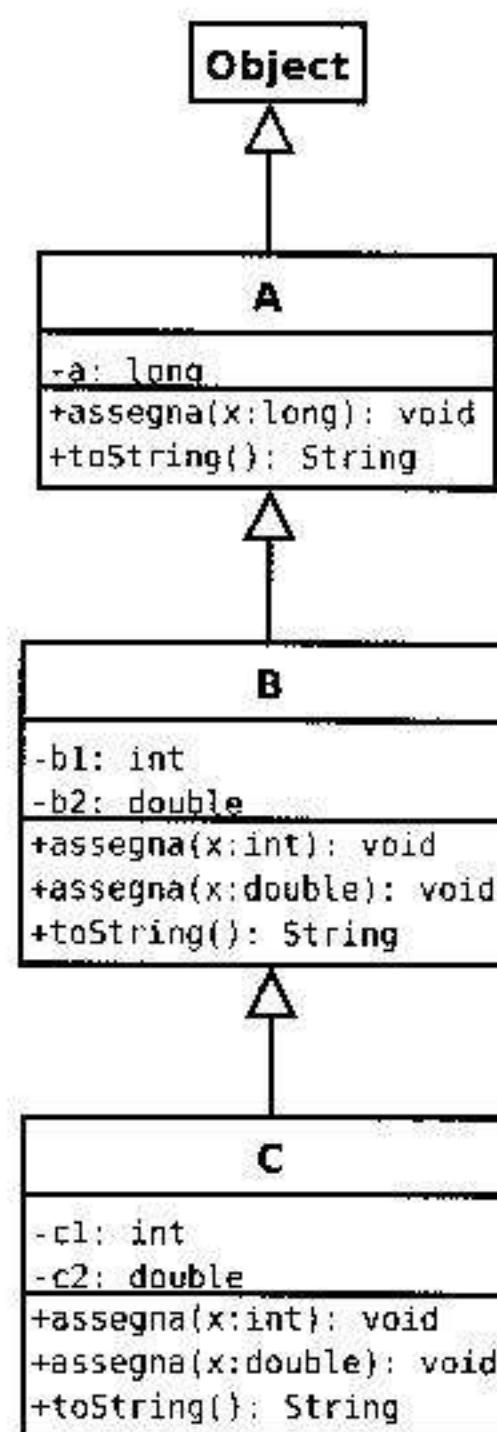


Figura 8.1 Gerarchia delle classi A, B e C.

Fase di compilazione: scelta della segnatura

Analizzando l'invocazione di un metodo abbiamo già evidenziato come il compilatore controlli che esista almeno un metodo invocabile per il tipo corrispondente al riferimento.

Ad esempio le chiamate `gamma.toString()` e `gamma.equals(alfa)` sono lecite in quanto il tipo di `gamma`, cioè la classe C, dispone di un metodo `toString` (definito localmente) senza parametri e di un metodo `equals` (ereditato da `Object`) con parametro di tipo `Object` (cui verrà convertito l'argomento `alfa` di tipo A). A causa dell'overloading, in talune situazioni potrebbero esserci più metodi utilizzabili, o meglio potrebbero esserci differenti signature (ad esempio se invochiamo il metodo `assegna`).

Vediamo in dettaglio le operazioni effettuate dal compilatore.

(1) Scelta delle signature "candidate".

Tra le signature di tutti i metodi con lo stesso nome di quello invocato, disponibili per la classe corrispondente al riferimento, il compilatore individua quelle che possono soddisfare la chiamata. Affinché una segnatura possa soddisfare la chiamata dev'essere:

- *compatibile* con gli argomenti utilizzati nella chiamata: il numero dei parametri nella segnatura e il numero degli argomenti utilizzati nella chiamata devono essere uguali; ogni argomento dev'essere di un tipo assegnabile al corrispondente parametro (Capitolo 4);
- *accessibile* al codice chiamante (essendo ad esempio dichiarata `public`).

Se non esistono signature candidate, il compilatore segnala un errore.

Ad esempio, nella chiamata

`alfa.assegna(2)`

il riferimento è di tipo A. L'unico metodo `assegna` disponibile per A è quello dichiarato in A stessa, che riceve un parametro di tipo `long`. Poiché l'argomento utilizzato nella chiamata può essere assegnato a un `long` (dopo una promozione), la segnatura

`assegna(long x)`

è una segnatura candidata (in questo caso l'unica).

Per la chiamata

`alfa.assegna(2.0)`

l'argomento è di tipo `double`: non esistono metodi di nome `assegna` per A che possano ricevere argomenti di tipo `double`. La chiamata non è quindi corretta, e pertanto non viene accettata dal compilatore.

Consideriamo invece la chiamata

`beta.assegna(2)`

In questo caso il riferimento è di tipo B. Per questo tipo ci sono tre segnature candidate, e cioè:

```
assegna(int x)
assegna(double x)
assegna(long x)
```

Le prime due segnature corrispondono a metodi dichiarati direttamente nella classe B, la terza a un metodo ereditato da A.

(2) *Scelta della segnatura “più specifica”.*

Tra tutte le segnature candidate, individuate nel passo precedente, il compilatore seleziona quella che più si avvicina alla chiamata, cioè quella che richiede il minor numero di promozioni. Nell'ultimo esempio, l'argomento utilizzato nella chiamata è di tipo `int`. La prima delle segnature selezionate è quindi la più adatta a questo tipo (non richiede alcuna promozione).⁸ Qualora il compilatore non riesca a individuare una segnatura più specifica delle altre, fornirà un messaggio d'errore. Un esempio di questa situazione si ha considerando una chiamata come (per semplicità omettiamo di indicare il riferimento):

```
z(1, 2)
```

nel caso siano disponibili solo due metodi `z` con le seguenti segnature:

```
z(double x, int y)
e
z(int x, double y)
```

Fase di esecuzione: scelta del metodo

Il metodo che dev'essere effettivamente eseguito viene selezionato al momento dell'esecuzione *in base al tipo dell'oggetto*. In particolare viene eseguito un metodo la cui segnatura sia *esattamente quella selezionata in fase di compilazione*. Si ricerca tale metodo risalendo la gerarchia delle classi a partire dalla classe corrispondente all'oggetto che deve eseguire il metodo.

Consideriamo, ad esempio, la seguente classe Prova:

```
import prog.io.*;
class Prova {
```

⁸ Una segnatura con k parametri di tipi T_1, T_2, \dots, T_k è *più specifica* di una segnatura con k parametri di tipi U_1, U_2, \dots, U_k se, per ogni indice i , il tipo T_i può essere assegnato al tipo U_i .

```

public static void main(String[] a) {
    ConsoleOutputManager out = new ConsoleOutputManager();

    C gamma = new C();
    A alfa = gamma;
    B beta = gamma;
    out.println(alfa.toString());
    alfa.assegna(3L);
    out.println(alfa.toString());
    alfa.assegna(2);
    out.println(alfa.toString());
    beta.assegna(4);
    out.println(alfa.toString());
    gamma.assegna(8d);
    out.println(alfa.toString());
}
}

```

Per ognuna delle chiamate, secondo quanto abbiamo spiegato sopra, il compilatore stabilisce la segnatura del metodo che dovrà essere eseguito.

Ecco il codice della classe riscritto annotando accanto a ogni chiamata di un metodo `assegna` un commento indicante la segnatura selezionata dal compilatore:

```

import prog.io.*;

class Prova {
    public static void main(String[] a) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        C gamma = new C();
        A alfa = gamma;
        B beta = gamma;
        out.println(alfa.toString());
        alfa.assegna(3L); //assegna(long x)
        out.println(alfa.toString());
        alfa.assegna(2); //assegna(long x)
        out.println(alfa.toString());
        beta.assegna(4); //assegna(int x)
        out.println(alfa.toString());
        gamma.assegna(8d); //assegna(double x)
        out.println(alfa.toString());
    }
}

```

Esaminiamo ora ciò che avviene in fase di esecuzione. Nella prima istruzione del metodo `main` c'è una chiamata al costruttore di `C`. Osserviamo che la classe `C` non contiene esplicitamente un costruttore. È pertanto disponibile il costruttore privo di argomenti che richiama implicitamente il costruttore privo di argomenti della superclasse `B`. Tale costruttore è invece scritto esplicitamente. Come prima operazione, esso richiama (implicitamente) il costruttore senza argomenti della superclasse `A`. Il costruttore di `A` (anche in questo caso non esplicito) costruisce un oggetto contenente il campo `long a`, inizializzato con il valore di default `0L`. Una volta costruito l'oggetto, il costruttore di `B` lo adatta aggiungendovi i campi `b1` e `b2` inizializzati a `1`. Infine il costruttore di `C` vi aggiunge i campi `c1` e `c2` inizializzati a zero. Inoltre, l'oggetto verrà marcato come istanza della classe `C`.

Lo stato in cui si trova inizialmente l'oggetto può essere dunque descritto, evidenziando anche i campi ereditati dalle superclassi, come segue:

Istanza di `C`:

Istanza di `B`:

Istanza di `A`:

```
a: 0L
b1: 1
b2: 1.0
c1: 0
c2: 0.0
```

Il riferimento al nuovo oggetto viene poi assegnato alla variabile `gamma` e, nelle istruzioni successive, alle variabili `alfa` e `beta`. Pertanto i tre riferimenti indicano il medesimo oggetto.

Per ottenere una stringa da visualizzare, viene successivamente richiamato il metodo `toString` tramite il riferimento `alfa`. Poiché l'oggetto riferito da `alfa` è un'istanza della classe `C`, sarà eseguito direttamente il metodo `toString` di `C`. Visualizzando il risultato si ottiene:

```
a = 0, b1 = 1, b2 = 1.0, c1 = 0, c2 = 0.0
```

La riga successiva contiene la chiamata

```
alfa.assegna(3L)
```

alla quale il compilatore ha associato la segnatura `assegna(long x)`. Perciò viene cercato un metodo che abbia esattamente tale segnatura a partire dalla classe effettiva dell'oggetto, cioè da `C`. Tale metodo non è presente né in `C` né in `B`, ma viene individuato in `A`. Quindi viene eseguito il metodo `assegna` di `A`, che modifica il valore del campo `a`. Lo stato dell'oggetto diventa pertanto:

Istanza di `C`:

Istanza di `B`:

Istanza di `A`:

```
a: 3L
b1: 1
b2: 1.0
c1: 0
c2: 0.0
```

e l'istruzione successiva visualizza:

```
a = 3, b1 = 1, b2 = 1.0, c1 = 0, c2 = 0.0
```

Nella successiva chiamata

```
alfa.assegna(2)
```

la segnatura scelzionata dal compilatore è ancora `assegna(long x)`. Con lo stesso procedimento si cerca un metodo con la segnatura selezionata a partire dalla classe dell'oggetto, cioè da C. Tale metodo viene trovato in A. La sua esecuzione modifica nuovamente il campo a, ottenendo nell'istruzione successiva il seguente output:

```
a = 2, b1 = 1, b2 = 1.0, c1 = 0, c2 = 0.0
```

La chiamata seguente è:

```
beta.assegna(4)
```

per la quale il compilatore ha selezionato la segnatura `assegna(int x)`. Nel corso dell'esecuzione viene ricercato un metodo con la stessa segnatura a partire dalla classe dell'oggetto, cioè da C. Esso viene immediatamente individuato in C. La sua esecuzione modifica il campo c1. Pertanto l'output prodotto dall'istruzione seguente è:

```
a = 2, b1 = 1, b2 = 1.0, c1 = 4, c2 = 0.0
```

Infine, per la chiamata

```
gamma.assegna(8d)
```

la segnatura selezionata dal compilatore è `assegna(double x)`. Anche in questo caso, viene direttamente individuato il metodo presente nella classe C, che modifica il campo c2. L'output prodotto dall'istruzione successiva è:

```
a = 2, b1 = 1, b2 = 1.0, c1 = 4, c2 = 8.0
```

8.7 Il metodo equals

Riconsideriamo il metodo `equals` della classe `Frazione` alla luce di quanto abbiamo appena visto. A tale scopo presentiamo una classe di prova che legge i dati relativi a due frazioni e controlla se esse rappresentano lo stesso valore numerico:

```
import prog.io.*;  
  
class Uguali {  
  
    public static void main(String args[]) {
```

```
ConsoleInputManager in = new ConsoleInputManager();
ConsoleOutputManager out = new ConsoleOutputManager();

int n, d;

out.println("Inserire i dati della prima frazione");
n = in.readInt(" numeratore? ");
d = in.readInt(" denominatore? ");
Frazione f1 = new Frazione(n, d);

out.println("Inserire i dati della seconda frazione");
n = in.readInt(" numeratore? ");
d = in.readInt(" denominatore? ");
Frazione f2 = new Frazione(n, d);

if (f1.equals(f2))
    out.println("Le due frazioni sono uguali");
else
    out.println("Le due frazioni sono diverse");
}
```

Ecco un esempio di esecuzione in cui vengono fornite due frazioni con lo stesso valore:

```
Inserire i dati della prima frazione
numeratore? 3
denominatore? 4
Inserire i dati della seconda frazione
numeratore? 6
denominatore? 8
Le due frazioni sono uguali
```

Segue un esempio in cui vengono inserite due frazioni con valori differenti:

```
Inserire i dati della prima frazione
numeratore? 1
denominatore? 4
Inserire i dati della seconda frazione
numeratore? 3
denominatore? 8
Le due frazioni sono diverse
```

Riportiamo il metodo `equals` definito nell'ultima versione della classe `Frazione` (Paragrafo 7.3). In tale versione, il costruttore memorizza la frazione in forma semplificata:

```
public boolean equals(Frazione f) {
    return this.num == f.num && this.den == f.den;
}
```

Non avendo indicato nulla nella sua intestazione, la classe **Frazione** è una sottoclasse diretta della classe **Object**. Ricordiamo che anche **Object** ha un proprio metodo **equals** che, ricevendo un parametro **Object**, restituisce **true** solo nel caso in cui l'oggetto che esegue il metodo coincida con l'oggetto di cui viene fornito il riferimento tramite il parametro. Il metodo **equals** di **Object** potrebbe dunque essere scritto come:

```
public boolean equals(Object altro) {
    return this == altro;
}
```

Osserviamo che i due metodi **equals** hanno segnatura differente. Quindi, c'è overloading e non overriding. In sostanza, la classe **Frazione** dispone di due metodi di nome **equals**:

- **public boolean equals(Frazione f)**
È il metodo **equals** definito nella classe **Frazione**.
- **public boolean equals(Object altro)**
È il metodo **equals** che la classe **Frazione** eredita dalla superclasse **Object**.

Nel metodo **main** della classe **Uguali**, per verificare l'uguaglianza delle frazioni all'interno dell'**if** si utilizza la condizione

```
f1.equals(f2)
```

Il riferimento **f1** all'oggetto che deve eseguire il metodo è di tipo **Frazione**: per esso, come abbiamo appena osservato, esistono due metodi **equals**. Poiché l'argomento **f2** è di tipo **Frazione**, il compilatore risolve l'overloading selezionando la segnatura del metodo **equals** definito nella classe **Frazione**, cioè quella con parametro **Frazione**. In questo caso, la chiamata

```
f1.equals(f2)
```

non presenta polimorfismo perché c'è un unico metodo **equals** con parametro di tipo **Frazione**.

Esaminiamo ora che cosa succede se dichiariamo ambedue i riferimenti **f1** e **f2** di tipo **Object**:

```
import prog.io.*;

class Uguali {

    public static void main(String args[]) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();
```

```

int n, d;

out.println("Inserire i dati della prima frazione");
n = in.readInt(" numeratore? ");
d = in.readInt(" denominatore? ");
Object f1 = new Frazione(n, d); //f1 è di tipo Object

out.println("Inserire i dati della seconda frazione");
n = in.readInt(" numeratore? ");
d = in.readInt(" denominatore? ");
Object f2 = new Frazione(n, d); //f2 è di tipo Object

if (f1.equals(f2))
    out.println("Le due frazioni sono uguali");
else
    out.println("Le due frazioni sono diverse");
}
}

```

Mandando in esecuzione il codice della classe e fornendo in ingresso due frazioni con il medesimo valore, la risposta sarà che le due frazioni sono diverse:

```

Inserire i dati della prima frazione
numeratore? 3
denominatore? 4
Inserire i dati della seconda frazione
numeratore? 6
denominatore? 8
Le due frazioni sono diverse

```

Vediamo di capire la ragione di questo comportamento, a prima vista sorprendente.

Analizzando l'istruzione

```
f1.equals(f2)
```

il compilatore esamina le segnature dei metodi `equals` disponibili per la classe del riferimento `f1`, che in questo caso è `Object` (e non `Frazione` come nel caso precedente). L'unico metodo `equals` disponibile è quindi quello che riceve un parametro di tipo `Object`. Dunque il compilatore seleziona la segnatura con parametro di tipo `Object`.

Come spiegato nel paragrafo precedente, durante l'esecuzione la Java Virtual Machine deve ricercare un metodo, con la segnatura stabilita dal compilatore, risalendo nella gerarchia delle classi a partire dalla classe effettiva dell'oggetto che esegue il metodo. In questo caso l'oggetto riferito da `f1` è di tipo `Frazione`. Nella classe `Frazione` non è definito un metodo `equals` con parametro `Object`. Pertanto la ricerca prosegue nella superclasse `Object`, dove invece il metodo con la segnatura cercata è presente. Tale metodo, come ricordato sopra, restituisce

`true` solo se l'oggetto che lo esegue coincide con quello passato tramite il parametro. Perciò, in questa versione dell'applicazione `Uguali`, il metodo `main` risponderà sempre che le frazioni sono diverse, in quanto gli oggetti cui si riferiscono `f1` e `f2` sono sempre distinti (perché creati con due `new` diverse) anche se potrebbero rappresentare lo stesso valore.

La medesima situazione si verifica se definiamo uno dei due riferimenti di tipo `Object` e l'altro di tipo `Frazione`. In particolare:

- Definendo `f1` di tipo `Object` e `f2` di tipo `Frazione`, il compilatore esamina le segnature dei metodi `equals` disponibili nella classe `Object` e trova come unico metodo quello che riceve il parametro `Object`. Di conseguenza durante l'esecuzione viene selezionato come prima il metodo ereditato da `Object`. Al momento della chiamata il riferimento `f2` sarà promosso al tipo `Object`.
- Definendo `f1` di tipo `Frazione` e `f2` di tipo `Object`, il compilatore esamina le segnature dei metodi `equals` disponibili nella classe `Frazione`: quello definito in `Frazione`, con parametro `Frazione`, e quello ereditato da `Object`, con parametro `Object`. In base al tipo dell'argomento viene anche in questo caso selezionato il metodo definito in `Object`.

Nelle situazioni che abbiamo appena analizzato c'è sempre overloading e non overriding: il metodo `equals` definito nella classe `Frazione` ha una segnatura *differente* dal metodo `equals` definito in `Object`. Pertanto il metodo `equals` di `Frazione` *non ridefinisce* quello di `Object`. C'è overriding, e dunque si sfrutta il polimorfismo solo quando un metodo di una sottoclasse ridefinisce un metodo della superclasse *utilizzando la stessa segnatura*.

In realtà, quando abbiamo scritto il metodo `equals` per la classe `Frazione`, la nostra intenzione era quella di “specializzare” il metodo `equals` di `Object` alle frazioni. In altre parole, avremmo voluto ridefinire il metodo `equals` di `Object` in modo tale che, nel caso di istanze di `Frazione`, fosse in grado di verificare l'uguaglianza di due frazioni. Per *ridefinire* un metodo è sempre *necessario* utilizzare la *stessa segnatura* del metodo che si vuole ridefinire. Pertanto nella classe `Frazione`, *in luogo* del metodo `equals` che abbiamo introdotto, avremmo dovuto scriverne uno con la seguente intestazione:

```
public boolean equals(Object altro)
```

D'altra parte in questa intestazione il parametro `altro` è di tipo `Object`. Di conseguenza il compilatore non permette di accedere ai campi `num` e `den` (infatti non è detto che una qualsiasi istanza di `Object` possieda tali campi). In sostanza, bisogna riscrivere il codice del metodo in modo che sia in grado di trattare un qualunque parametro `Object`.

Ricordiamo che il metodo che vogliamo scrivere appartiene alla classe `Frazione` e perciò sarà eseguito da un oggetto della classe, che deve verificare l'uguaglianza con un oggetto passato come parametro. L'uguaglianza può essere decisa utilizzando la seguente strategia:

```
if (altro è proprio una frazione)
    controlla l'uguaglianza tra le due frazioni
else
    i due oggetti non sono uguali
```

Per verificare che altro sia una frazione possiamo servirci dell'operatore `instanceof`. In caso positivo, si può forzare il riferimento altro al tipo `Frazione` mediante un cast e, a questo punto, effettuare il confronto accedendo ai campi `num` e `den`:

```
public boolean equals(Object altro) {
    if (altro instanceof Frazione) {
        Frazione a = (Frazione) altro;
        return this.num == a.num && this.den == a.den;
    }
    else
        return false;
}
```

Sostituendo ora nel testo della classe `Frazione` il metodo `equals` dato precedentemente con questo nuovo metodo, si *ridefinisce* il metodo `equals` di `Object`. Nel metodo `main` della versione della classe `Uguali`, con `f1` dichiarato di tipo `Object`, a questo punto c'è polimorfismo. Al momento della compilazione viene selezionata la segnatura di un metodo `equals` con parametro `Object` (l'unica disponibile). Al momento dell'esecuzione, poiché l'oggetto riferito da `f1` è stato costruito dalla classe `Frazione`, il metodo `equals` con parametro `Object` da eseguire sarà subito trovato nella classe `Frazione` (si tratta appunto del metodo definito qui sopra).

In alternativa, una soluzione equivalente, ma più elegante, consiste nello scrivere nella classe `Frazione` due metodi `equals`: uno con parametro `Object` e l'altro con parametro `Frazione`. Quest'ultimo è in grado di confrontare solo due frazioni. Il primo controlla invece il tipo dell'oggetto fornito tramite il parametro: se esso è una frazione, delega il proprio compito all'altro metodo `equals`:

```
public boolean equals(Object altro) {
    if (altro instanceof Frazione) {
        Frazione a = (Frazione) altro;
        return equals(a);
    } else
        return false;
}

public boolean equals(Frazione f) {
    return this.num == f.num && this.den == f.den;
}
```

8.8 Variabili e adombramento

Oltre a esserci metodi con la stessa segnatura, una classe e una sua sottoclasse potrebbero avere campi con lo stesso nome. Si considerino ad esempio le seguenti classi:

```

public class Sopra {
    int k = 1;

    public String toString() {
        return String.valueOf(k);
    }
}

public class Sotto extends Sopra {
    int k = 2;

    public String toString() {
        return k + ", " + super.toString();
    }
}

```

Ambedue le classi contengono un campo di nome k. Inoltre la classe Sotto è una sottoclasse della classe Sopra. Un'istanza di Sopra ha un'unica variabile denominata k (inizializzata a 1). Al contrario, un'istanza di Sotto ha due variabili denominate k: la prima, inizializzata a 1, ereditata dalla superclasse; la seconda, inizializzata a 2, corrispondente alla dichiarazione data nella classe Sotto, come evidenziato di seguito:

Istanza di Sotto:

Istanza di Sopra:

k: 1

k: 2

Per determinare quale variabile sarà utilizzata in un'istruzione, è sufficiente osservare il contesto in cui si trova l'istruzione nel testo della classe. In particolare l'istruzione `return String.valueOf(k)` del metodo `toString` della classe Sopra utilizza la variabile k della classe Sopra (anche nel caso in cui l'oggetto in esecuzione sia un'istanza della classe Sotto). Viceversa, l'istruzione `return k + ", " + super.toString()` nella classe Sotto utilizza la variabile k della classe Sotto. In questa situazione si dice che la variabile k *adombra*, nella classe Sotto, la variabile k della classe Sopra.

Consideriamo a questo punto la seguente classe di prova:

```

import prog.io.*;

class Prova {
    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        Sopra s;
        s = new Sopra();
    }
}

```

```

    out.println(s.toString());
    s = new Sotto();
    out.println(s.toString());
}
}

```

La prima espressione `new` richiama il costruttore della classe `Sopra`. Di conseguenza viene costruito un oggetto che ha un unico campo `k`, che contiene il valore 1. Alla variabile `s` viene assegnato il riferimento all'oggetto appena creato. L'istruzione successiva richiede all'oggetto riferito da `s` di eseguire il proprio metodo `toString`. Poiché l'oggetto è della classe `Sopra`, viene eseguito il metodo `toString` in tale classe: esso restituisce la stringa che rappresenta il valore del campo `k`, cioè 1.

Nell'istruzione successiva c'è di nuovo un'espressione `new`, questa volta seguita dal costruttore della classe `Sotto`. Pertanto viene creato un oggetto con due campi: il primo, denominato `k`, con valore 1, ereditato dalla classe `Sopra`; il secondo, sempre denominato `k`, ma con valore 2, corrispondente alla dichiarazione data in `Sotto`. Nel codice della classe `Sotto` il nuovo campo `k` adombra quello della classe `Sopra`. Il riferimento al nuovo oggetto è assegnato alla variabile `s` (di tipo `Sopra`). Nell'ultima istruzione si chiede all'oggetto riferito da `s` di eseguire il proprio metodo `toString`. Poiché l'oggetto è della classe `Sotto`, viene eseguito il metodo `toString` di tale classe (nonostante il riferimento sia di tipo `Sopra`). Tale metodo restituisce la stringa che rappresenta il valore del campo `k`, cioè 2, concatenata con la stringa restituita dal metodo `toString` della superclasse, che invece rappresenta il valore del campo `k` visibile nella superclasse, cioè 1.

In realtà, dalla sottoclasse si può accedere ai campi della superclasse utilizzando il riferimento `super` (purché essi non siano stati dichiarati `private`). Di conseguenza, nel codice della classe `Sotto` si potrebbe accedere al campo `k` della classe `Sopra` scrivendo `super.k`. Per esempio, il metodo `toString` della classe `Sotto` potrebbe essere scritto nel modo seguente:

```

public String toString() {
    return k + ", " + super.k;
}

```

Può esserci adombramento anche in altre situazioni. Ad esempio, in un metodo o in un costruttore, un parametro oppure una variabile locale possono adombrire un campo con lo stesso nome. Consideriamo la classe `Orario` del capitolo precedente. Per evidenziare il significato delle informazioni memorizzate nei campi abbiamo scelto per essi i nomi `ore` e `min`. A questo punto, proprio per evitare di incorrere in inconvenienti dovuti all'adombramento, abbiamo definito il costruttore (con due argomenti) della classe come segue:

```

public Orario(int hh, int mm) {
    ore = hh;
    min = mm;
}

```

In realtà, per rendere ancora più chiaro il significato dei parametri, anziché `hh` e `mm`, sarebbe stato comodo chiamarli direttamente `ore` e `min`:

```
public Orario(int ore, int min) {
    ...
}
```

In questo modo avremmo però adombrato, all'interno del costruttore, i campi della classe: con questa scrittura, i nomi `ore` e `min` indicano infatti all'interno del costruttore i parametri, e non i campi. È tuttavia possibile accedere ancora ai campi dell'oggetto che il costruttore sta inizializzando mediante il riferimento `this`, scrivendo il costruttore così:

```
public Orario(int ore, int min) {
    this.ore = ore;
    this.min = min;
}
```

Non ci addentreremo in ulteriori dettagli relativi a questo argomento, ma ricordiamo solamente che, in caso di omonimia, la variabile effettivamente visibile è quella la cui dichiarazione si trova “più vicina” all’istruzione che la utilizza. Si tenga comunque presente che è sempre possibile accedere a un campo utilizzando i riferimenti `this` o `super`.

8.9 Esempio

Consideriamo le seguenti classi A e B, rappresentate anche nella Figura 7.2:

```
public class A {
    int a;

    public void assegna(int x) {
        a = x;
    }

    public String toString() {
        return String.valueOf(a);
    }
}

public class B extends A {
    int b;

    public void assegna(int y, int z) {
        assegna(y);
        b = z;
    }
}
```

```

    }

public String toString() {
    return super.toString() + ", " + b;
}
}

```

La classe A estende Object. Poiché nel corpo della classe non è indicato alcun costruttore, la classe possiede il costruttore di default privo di argomenti che crea l'oggetto, con la variabile di istanza a inizializzata al valore di default 0 (in realtà questo costruttore chiama a sua volta il costruttore privo di argomenti della superclasse Object).

La classe ha due metodi: `assegna`, che riceve un valore di tipo int e lo assegna al campo a, e `toString`, che restituisce la stringa che rappresenta il contenuto del campo a.

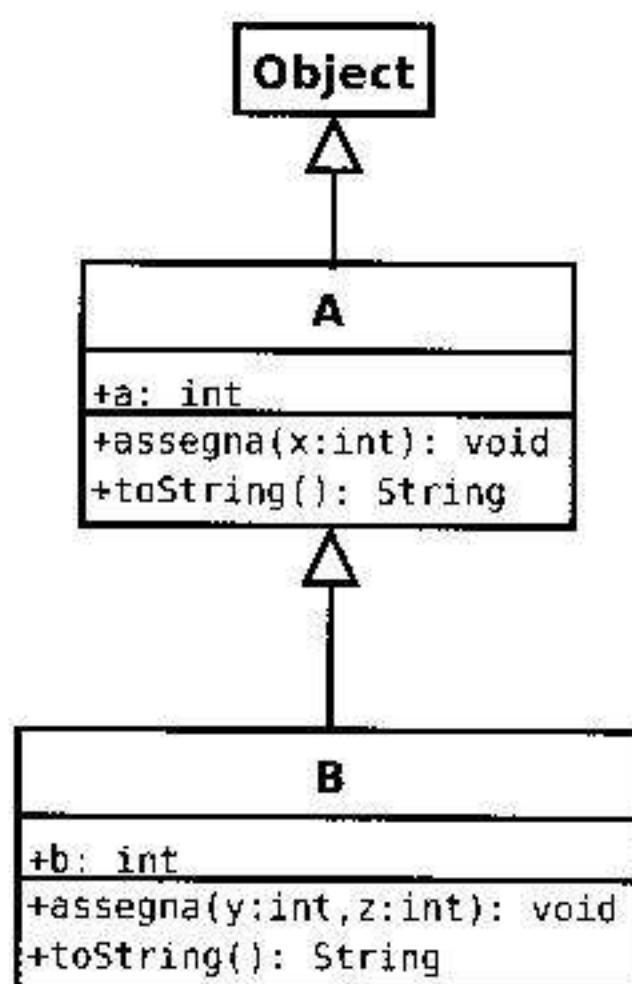


Figura 8.2 Gerarchia delle classi A e B.

Esaminiamo ora la classe B. Essa estende A e, indirettamente, Object. Un oggetto della classe B ha in più, rispetto agli oggetti della classe A, il campo b. Anche nella classe B non è definito esplicitamente un costruttore. Dunque B dispone del costruttore di default privo di argomenti. Questo costruttore lavora nel modo seguente: prima chiede alla superclasse di fabbricare un proprio oggetto (quindi un oggetto con il solo campo a), poi lo modifica aggiungendovi il campo b. Anche b viene inizializzato al valore di default che, per il tipo int, è zero.

La classe B dispone di due metodi; come per la classe A, essi si chiamano `assegna` e `toString`. Il metodo `assegna` di B ha una *segnatura differente* da quella del metodo `assegna` di A. Osserviamo che nel suo corpo viene richiamato il metodo `assegna` con un solo parametro. In questo caso non c'è ambiguità: il metodo con un parametro è sicuramente quello della superclasse.

In luogo di:

```
assegna(y);
```

per indicare che ci si riferisce al metodo della superclasse si sarebbe potuto scrivere:

```
super.assegna(y);
```

Nel caso del metodo `toString` la situazione è completamente diversa, perché i metodi `toString` di ambedue le classi hanno la *stessa segnatura*. In questo caso, il metodo `toString` di B invoca il metodo `toString` della superclasse mediante la chiamata

```
super.toString()
```

e al rientro aggiunge alla fine della stringa il valore della variabile `b`. Inoltre, l'omissione del riferimento `super` davanti al nome del metodo `toString` provocherebbe la chiamata del metodo `toString` di B (abbiamo detto, infatti, che la ricerca del metodo da eseguire avviene risalendo la gerarchia delle classi, a partire dalla classe cui appartiene l'oggetto che deve eseguire il metodo e che, in mancanza di altre indicazioni, è l'oggetto indicato dal riferimento `this`). In altre parole, il metodo `toString` richiamerebbe se stesso (chiamata *ricorsiva*), provocando in questa situazione un comportamento anomalo.

Ecco una classe di prova che utilizza la classe B:

```
import prog.io.*;

class Prova {
    public static void main(String[] args) {
        //predisposizione del canale di comunicazione
        ConsoleOutputManager out = new ConsoleOutputManager();

        B rif = new B();
        out.println(rif.toString());
        rif.assegna(1);
        out.println(rif.toString());
        rif.assegna(2, 3);
        out.println(rif.toString());
    }
}
```

L'output prodotto è:

```
0, 0
1, 0
2, 3
```

Esercizi

8.3 Considerate la seguente classe:

```
public class C extends B {
    int b;

    public void assegna(int p, int q, int r) {
        assegna(p, q);
        b = r;
    }

    public String toString() {
        return super.toString() + ", " + b;
    }
}
```

dove B è la classe presentata nell'esempio a pagina 378. Indicate tutte le superclassi di C. Considerate la seguente classe di prova, che utilizza C:

```
import prog.io.*;

class Prova {
    public static void main(String[] args) {
        //predisposizione del canale di comunicazione
        ConsoleOutputManager out = new ConsoleOutputManager();

        C rif = new C();
        out.println(rif.toString());
        rif.assegna(1);
        out.println(rif.toString());
        rif.assegna(2, 3);
        out.println(rif.toString());
        rif.assegna(4, 5, 6);
        out.println(rif.toString());
    }
}
```

Individuate l'output prodotto dal metodo main. Verificate la risposta sulla vostra macchina. Modificate ora la classe C sostituendo la chiamata super.toString() con toString(). Ricompilate quindi la classe C e la classe Prova, e fate eseguire il metodo main. Come si comporta la Java Virtual Machine? Fornite una spiegazione.

8.4 Definite un metodo equals con parametro Object per la classe Quadrato.

8.5 Riscrivete il metodo `equals` della classe `Frazione` in modo che possa ricevere un parametro della classe `Object`.

8.10 Implementazione della classe Figura

Nel Paragrafo 6.5 abbiamo utilizzato la classe `Figura`. A differenza delle classi di cui abbiamo presentato l'implementazione finora, si tratta di una *classe astratta*, e dunque non può essere istanziata. In particolare ricordiamo che una classe astratta può contenere *metodi astratti*, cioè metodi di cui viene fornita solo l'intestazione, ma non l'implementazione, che dovrà essere obbligatoriamente data nelle sottoclassi concrete.

Nella classe `Figura` vogliamo fornire i metodi `getArea` e `getPerimetro` per il calcolo, rispettivamente, dell'area e del perimetro della figura rappresentata dall'oggetto che esegue il metodo, e i metodi `haAreaMaggiore` e `haPerimetroMaggiore` per il confronto dell'area e del perimetro della figura rappresentata dall'oggetto con l'area e il perimetro di una figura di cui viene fornito il riferimento tramite il parametro.

Questi ultimi due metodi possono essere codificati senza difficoltà richiamando i primi due. A titolo d'esempio discutiamo l'implementazione del metodo `haAreaMaggiore`, che avrà l'intestazione:

```
public boolean haAreaMaggiore(Figura altra)
```

e dovrà restituire `true` quando l'area della figura che esegue il metodo è maggiore di quella della figura fornita tramite il parametro.

Il corpo del metodo potrà essere codificato secondo questo schema:

```
calcola l'area della figura che esegue il metodo
calcola l'area della figura fornita tramite il parametro
if (prima area > seconda area)
    return true;
else
    return false;
```

Per il calcolo dell'area di una figura si può richiedere alla figura stessa di eseguire il proprio metodo `getArea`. Di conseguenza il metodo può essere codificato seguendo lo schema precedente come:

```
public boolean haAreaMaggiore(Figura altra) {
    double area1 = this.getArea();
    double area2 = altra.getArea();
    if (area1 > area2)
        return true;
    else
        return false;
}
```

Il metodo può essere scritto in maniera più compatta come:

```
public boolean haAreaMaggiore(Figura altra) {
    return this.getArea() > altra.getArea();
}
```

Il calcolo dell'area e del perimetro dipende invece dal tipo di figura considerata. Pertanto non è possibile fornire l'implementazione dei metodi `getArea` e `getPerimetro` direttamente nella classe `Figura`. Ci limitiamo quindi a indicarne l'intestazione specificando che si tratta di metodi astratti. Ad esempio, per il metodo `getArea` ci limitiamo a scrivere:

```
public abstract double getArea();
```

Non specifichiamo alcun costruttore all'interno del corpo della classe. Sarà dunque implicitamente inserito il costruttore privo di argomenti. Ecco il codice completo della classe:

```
public abstract class Figura {

    public abstract double getArea();

    public abstract double getPerimetro();

    public boolean haAreaMaggiore(Figura altra) {
        return this.getArea() > altra.getArea();
    }

    public boolean haPerimetroMaggiore(Figura altra) {
        return this.getPerimetro() > altra.getPerimetro();
    }

}
```

Si noti la parola `abstract` nell'intestazione per indicare che la classe è astratta. Inoltre, non essendo utilizzata la clausola `extends`, la classe è una sottoclasse diretta di `Object` e quindi ne eredita i metodi `toString` ed `equals`. Non ha molto senso ridefinire questi metodi all'interno della classe `Figura`, ma sarà opportuno ridefinirli nelle sue sottoclassi concrete.

Osserviamo che, sebbene la classe `Figura` sia priva di campi, generalmente le classi astratte possono averne.

8.11 Implementazione della classe Rettangolo

Presentiamo ora una possibile implementazione della classe `Rettangolo`. Prima di tutto la classe è una sottoclasse concreta di `Figura`. Pertanto la sua intestazione sarà:

```
public class Rettangolo extends Figura
```

Vediamo ora come definire lo stato di un oggetto della classe. Gli oggetti che vogliamo rappresentare sono rettangoli. Ognuno di essi è caratterizzato da due grandezze: la lunghezza della base e quella dell'altezza. Dunque, una rappresentazione estremamente naturale si basa su due campi, che chiameremo appunto `base` e `altezza`. Siccome le grandezze rappresentate possono essere numeri reali, decidiamo di dichiarare i campi di tipo `double`. Inoltre, volendo nascondere l'implementazione della classe, decidiamo di dichiarare tali campi `private`. All'interno del corpo della classe scriveremo quindi la dichiarazione:

```
private double base, altezza;
```

Dunque un rettangolo, oltre a essere una figura, possiede due campi `base` e `altezza`. Analogamente a quanto fatto per la classe `Frazione`, possiamo facilmente scrivere il costruttore che si limita ad assegnare ai due campi i valori forniti tramite i parametri (ricordiamo che comunque il costruttore richiamerà implicitamente quello privo di argomenti della superclasse `Figura`):

```
public Rettangolo (double x, double y) {  
    base = x;  
    altezza = y;  
}
```

Anche gli altri metodi possono essere codificati facilmente. Sottolineiamo in particolare il metodo `equals`, con parametro `Object`, che ridefinisce quello ereditato dalla superclasse (e da `Object`). Per la codifica del metodo seguiamo esattamente lo schema utilizzato per il metodo `equals` della classe `Frazione` (Paragrafo 8.7).

```
public class Rettangolo extends Figura {  
    //CAMPI  
    private double base, altezza;  
  
    //COSTRUTTORI  
    public Rettangolo (double x, double y) {  
        base = x;  
        altezza = y;  
    }  
  
    //METODI  
    public double getArea() {  
        return base * altezza;  
    }  
  
    public double getPerimetro() {  
        return 2 * (base + altezza);  
    }  
  
    public double getAltezza() {
```

```

        return altezza;
    }

    public double getBase() {
        return base;
    }

    public String toString() {
        return "base = " + base + ", altezza = " + altezza;
    }

    public boolean equals(Rettangolo altro) {
        return this.base == altro.base && this.altezza == altro.altezza;
    }

    public boolean equals(Object o) {
        if (o instanceof Rettangolo)
            return equals((Rettangolo) o);
        else
            return false;
    }
}

```

La codifica della classe Quadrato presentata nel Paragrafo 8.1 è del tutto indipendente dall'implementazione data per Rettangolo. Se si modifica l'implementazione di Rettangolo rispettando i prototipi e i contratti dei metodi, non occorre alcuna modifica all'implementazione di Quadrato. In particolare, avendo definito *private* i campi *base* e *altezza*, essi non sono accessibili nemmeno dal codice della sottoclassificazione Quadrato.

All'interno della classe Quadrato non abbiamo definito alcun metodo *equals*. Di conseguenza la classe erediterà tali metodi da Rettangolo. Che cosa succede se tentiamo di confrontare un quadrato con un rettangolo? Se le due figure sono geometricamente differenti, l'uso dei metodi ereditati da Rettangolo indica, correttamente, che le figure non sono uguali. Ma che cosa succede eseguendo, ad esempio, il seguente frammento di codice?

```

Rettangolo r = new Rettangolo(4, 4);
Quadrato q = new Quadrato(4);
if (q.equals(r))
    out.println("uguali");
else
    out.println("diversi");

```

Dal punto di vista della gerarchia delle classi, gli oggetti riferiti da *r* e da *q* sono diversi. Tuttavia dal punto di vista geometrico essi rappresentano esattamente la stessa figura. Possiamo osservare

che questa porzione di codice fornirà la stringa "uguali", privilegiando quindi il punto di vista geometrico. In sostanza il tipo di confronto definito per i rettangoli va bene anche per i quadrati: non è perciò necessario specializzare o ridefinire il metodo `equals` nella sottoclasse.

Esercizi

- 8.6 Scrivete un'implementazione della classe `Rettangolo` in cui si utilizzano due campi di tipo `double`, che corrispondono, rispettivamente, all'area e alla base del rettangolo rappresentato.
- 8.7 Si può scrivere un'implementazione di `Rettangolo` in cui gli oggetti vengono rappresentati con due campi di tipo `double` che corrispondono, rispettivamente, all'area e al perimetro del rettangolo?
- 8.8 Inventate altre implementazioni di `Rettangolo`.
- 8.9 Scrivete un'implementazione della classe `Cerchio` (il valore della costante π è disponibile come campo statico denominato `PI` nella classe `Math`).
- 8.10 Scrivete una sottoclasse `Triangolo` di `Figura`; scrivete poi due sottoclassi `Triangolo-Equilatero` e `TriangoloRettangolo` di `Triangolo`.

8.12 Nuovi metodi per le classi `Rettangolo` e `Quadrato`

Nelle classi che abbiamo implementato non esistono metodi che permettano di modificare lo stato di un oggetto. Ad esempio, una volta che un oggetto di tipo `Rettangolo` è stato costruito, non è possibile modificarne la base o l'altezza.

Non è difficile aggiungere alla classe metodi che compiano queste operazioni. Poiché questi nuovi metodi verranno ereditati dalle sottoclassi, in questo caso da `Quadrato`, occorre fare molta attenzione per evitare che gli oggetti possano raggiungere stati inconsistenti. In questo paragrafo presentiamo e discutiamo tale problema.

Iniziamo col codificare un nuovo metodo di nome `cambiaBase`, da aggiungere alla classe `Rettangolo`. Questo metodo riceve un argomento di tipo `double`: il nuovo valore della base del rettangolo. Il metodo compie un'operazione interna e non restituisce alcun risultato. Ciò viene indicato utilizzando nell'intestazione del metodo la parola riservata `void` in luogo del tipo restituito:

```
public void cambiaBase(double x)
```

Nel corpo del metodo è sufficiente assegnare il valore del parametro `x` al campo `base` dell'oggetto scrivendo:

```
this.base = x;
```

o più semplicemente, non essendovi ambiguità:

```
base = x;
```

Per rientrare dal metodo si utilizza infine l'istruzione `return`. Dato che il metodo non deve restituire nulla, l'istruzione non è seguita da alcuna espressione. Ecco il testo del metodo:

```
public void cambiaBase(double x) {  
    base = x;  
    return;  
}
```

Dal momento che è l'ultima del metodo e che questo non deve restituire alcun risultato, l'istruzione `return` può anche essere omessa.

Ecco il testo completo della nuova versione della classe `Rettangolo` alla quale, oltre al metodo `cambiaBase`, abbiamo aggiunto un analogo metodo `cambiaAltezza`:

```
public class Rettangolo extends Figura {  
    //CAMPI  
    private double base, altezza;  
  
    //COSTRUTTORI  
    public Rettangolo(double x, double y) {  
        base = x;  
        altezza = y;  
    }  
  
    //METODI  
    public double getArea() {  
        return base * altezza;  
    }  
  
    public double getPerimetro() {  
        return 2 * (base + altezza);  
    }  
  
    public double getAltezza() {  
        return altezza;  
    }  
  
    public double getBase() {  
        return base;  
    }  
  
    public String toString() {
```

```
    return "base = " + base + ", altezza = " + altezza;
}

public boolean equals(Rettangolo altro) {
    return this.base == altro.base && this.altezza == altro.altezza;
}

public boolean equals(Object o) {
    if (o instanceof Rettangolo)
        return equals((Rettangolo) o);
    else
        return false;
}

public void cambiaBase(double x) {
    base = x;
}

public void cambiaAltezza(double x) {
    altezza = x;
}

}
```

Si invita il lettore a sviluppare un'applicazione di prova per la nuova versione della classe *Rettangolo*, nella quale, dopo avere letto i dati relativi a un rettangolo, si fornisce all'utente la possibilità di modificarlo.

Analogamente a quanto abbiamo appena fatto per la classe *Rettangolo*, aggiungiamo anche alla classe *Quadrato* un metodo, che chiameremo *cambiaLato*, che permetta di modificare il lato del quadrato rappresentato dall'oggetto. Ricordandoci che un quadrato è un rettangolo con base e altezza uguali, possiamo codificare il metodo (ignorando anche in questo caso l'implementazione della classe *Rettangolo* e servendoci solo dei metodi che essa offre) come:

```
public void cambiaLato(double x) {
    cambiaBase(x);
    cambiaAltezza(x);
}
```

Anche in questo caso si invita il lettore a sviluppare un'applicazione di prova per la nuova versione della classe *Quadrato*.

Osserviamo ora che, in base al meccanismo dell'ereditarietà descritto in questo capitolo, la classe *Quadrato* eredita da *Rettangolo* anche i metodi *cambiaBase* e *cambiaAltezza*. Dunque possiamo chiedere a un oggetto di tipo *Quadrato* di eseguire uno di questi metodi. In

questo modo potremmo portare l'oggetto in uno stato inconsistente. Infatti, se q è un riferimento di tipo Quadrato, l'istruzione

```
q.cambiaBase(6);
```

avrebbe l'effetto di modificare solo il campo base dell'oggetto a cui fa riferimento q; a questo punto invocazioni dei metodi `getArea` e `getPerimetro` restituirebbero valori scorretti.

Evidentemente l'operazione `cambiaBase`, pensata per i rettangoli, è inadeguata agli oggetti di tipo Quadrato, in quanto ne "corrompe" lo stato o, in altri termini, fa venire meno la proprietà che li caratterizza rispetto a quelli di tipo Rettangolo: il fatto, cioè, di avere la base uguale all'altezza.

Questo problema può essere risolto stabilendo che nel caso degli oggetti di tipo Quadrato i significati di `cambiaBase` e `cambiaAltezza` siano gli stessi di `cambiaLato`. In altre parole, chiedere a un quadrato di cambiare la propria base o di cambiare la propria altezza equivale a chiedergli di cambiare il proprio lato. Pertanto i metodi `cambiaBase` e `cambiaAltezza` possono essere ridefiniti nella classe Quadrato come:

```
public void cambiaBase(double x) {
    cambiaLato(x);
}

public void cambiaAltezza(double x) {
    cambiaLato(x);
}
```

A questo punto, inserendo i due metodi in Quadrato otteniamo la nuova versione (scorretta!):

```
public class Quadrato extends Rettangolo {
    //COSTRUTTORI
    public Quadrato(double x) {
        super(x, x);
    }

    //METODI
    public double getLato() {
        return getBase();
    }

    public void cambiaLato(double x) {
        cambiaBase(x);           //scorretto
        cambiaAltezza(x);        //scorretto
    }

    public String toString() {
```

```

    return "lato = " + getBase();
}

public void cambiaBase(double x) {
    cambiaLato(x);
}

public void cambiaAltezza(double x) {
    cambiaLato(x);
}

}

```

In particolare il metodo `cambiaLato`, che rispetto alla versione precedente *non* è stato modificato, non è più corretto. Ricordiamo che quando in un'invocazione di metodo non viene esplicitato il riferimento a un oggetto, è sempre sottinteso il riferimento `this` all'oggetto che esegue il metodo. Pertanto il metodo `cambiaLato` richiama i metodi `cambiaBase` e `cambiaAltezza` lasciando sottinteso il riferimento `this`. Nella versione precedente questi metodi non erano definiti nella classe `Quadrato`, e pertanto venivano cercati, e trovati, nella superclasse `Rettangolo` dalla quale erano ereditati. In questo caso, invece, sono disponibili nella classe `Quadrato` due metodi con i nomi specificati: pertanto verranno invocati i metodi `cambiaBase` e `cambiaAltezza` presenti in `Quadrato`, che a loro volta richiameranno `cambiaLato`, etc. In sostanza si crea una sequenza infinita di chiamate. Poiché ogni chiamata di metodo richiede l'occupazione di uno spazio di memoria, ciò provocherà l'esaurimento della memoria disponibile, e quindi un errore in esecuzione (si consiglia di effettuare un esperimento invocando il metodo `cambiaLato` in un'applicazione che utilizzi questa versione scorretta di `Quadrato`).

Per correggere il codice ripensiamo a come abbiamo implementato il metodo `cambiaLato`: in questo metodo immaginiamo di vedere la figura come un rettangolo e di cambiargli prima la base e poi l'altezza. A questo scopo dobbiamo servirci dei metodi `cambiaBase` e `cambiaAltezza` della superclasse `Rettangolo` tramite il riferimento `super` prima del nome del metodo:

```

public void cambiaLato(int x) {
    super.cambiaBase(x);
    super.cambiaAltezza(x);
}

```

Ecco il codice corretto della nuova versione della classe `Quadrato`:

```

public class Quadrato extends Rettangolo {
    //COSTRUTTORI
    public Quadrato(double x) {
        super(x, x);
    }
}

```

```
//METODI
public double getLato() {
    return getBase();
}

public String toString() {
    return "lato = " + getLato();
}

public void cambiaLato(double x) {
    super.cambiaBase(x);
    super.cambiaAltezza(x);
}

public void cambiaBase(double x) {
    cambiaLato(x);
}

public void cambiaAltezza(double x) {
    cambiaLato(x);
}

}
```

8.13 Il modificatore di visibilità protected

Nel capitolo precedente abbiamo spiegato il significato dei modificatori di visibilità `public`, `private` e amichevole. L'ultimo modificatore di visibilità che dobbiamo ancora considerare è `protected`, la cui applicazione è legata all'ereditarietà. All'interno di un package il modificatore `protected` equivale al modificatore amichevole. Un membro di una classe dichiarato `protected` è quindi visibile all'interno del codice di tutte le classi che compaiono nel medesimo package. All'esterno del package i membri `protected` di una classe sono invece visibili solo alle classi che la estendono direttamente o indirettamente, e solo a certe condizioni che descriveremo negli esempi seguenti.

Consideriamo questa classe definita in un package di nome `pack4`:

```
package pack4;

public class ClasseBase {
    protected int x;

    protected ClasseBase (int x) {
```

```
    this.x = x;
}

protected String visibileAlleSottoclassi() {
    return "pack4.ClasseBase: metodo visibileAlleSottoclassi";
}
}
```

Questa classe dichiara `protected` il campo `x`, il metodo `visibileAlleSottoclassi` e il costruttore con un argomento. Come abbiamo detto, possiamo utilizzarli all'interno del package come se avessero visibilità amichevole. Ad esempio, la seguente classe `ProvaDentro.java`, definita anch'essa nel package `pack4`, li utilizza liberamente.

```
package pack4;

import prog.io.*;

public class ProvaDentro {
    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        ClasseBase b = new ClasseBase(3);
        out.println("x = " + b.x);
        out.println(b.visibileAlleSottoclassi());
    }
}
```

Al contrario non viene compilata la classe:

```
import prog.io.*;
import pack4.*;

class Prova {

    public static void main(String[] args) {
        ConsoleOutputManager out = new ConsoleOutputManager();

        ClasseBase b = new ClasseBase(3);
        out.println("x = " + b.x);
        out.println(b.visibileAlleSottoclassi());
    }
}
```

Il compilatore segnala in particolare i seguenti errori:

```

Prova.java:9: ClasseBase(int) has protected access in pack4.ClasseBase
    ClasseBase b = new ClasseBase(3);
               ^
Prova.java:10: x has protected access in pack4.ClasseBase
    out.println("x = " + b.x);
               ^
Prova.java:11: visibileAlleSottoclassi() has protected access
in pack4.ClasseBase
    out.println(b.visibileAlleSottoclassi());
               ^
3 errors

```

Osserviamo che i messaggi di errore segnalano chiaramente che il costruttore con due argomenti e i membri `x` e `visibileAlleSottoclassi`, pur esistendo, non sono accessibili per via della visibilità attribuita loro: all'esterno del package in cui la classe è definita, l'accesso ai membri e ai costruttori `protected` è consentito esclusivamente alle sottoclassi. Ci sono inoltre le seguenti restrizioni (sempre nel caso di accesso dall'esterno del package):

- *Costruttori*

È possibile invocare un costruttore `protected` di una classe esclusivamente nel codice dei costruttori delle classi che la estendono direttamente tramite l'invocazione di `super`.

- *Campi e metodi*

I campi e i metodi `protected` di una classe A sono accessibili solo nel corpo delle classi B che la estendono (direttamente o indirettamente) ed esclusivamente tramite variabili riferimento o espressioni il cui tipo sia tutt'alpiù quello della sottoclasse B.

Consideriamo ad esempio la classe

```

import prog.io.*;

class ClasseEstesa extends pack4.ClasseBase {

    public ClasseEstesa(int x) {
        super(x);
    }

    public void usaProtected() {
        ConsoleOutputManager out = new ConsoleOutputManager();
        out.println("x = " + x);
        out.println(visibileAlleSottoclassi());
    }

    public static void main (String[] args) {

```

```

    ClasseEstesa e = new ClasseEstesa(3);
    e.usaProtected();
}
}

```

In questo caso il costruttore `protected` è utilizzato correttamente (è invocato tramite `super` nel costruttore della sottoclasse). Anche il campo e il metodo `protected` sono utilizzati correttamente. Infatti, sono impiegati all'interno del codice di una classe che estende `ClasseBase` (per la precisione nel metodo `usaProtected`) e sono utilizzati tramite un riferimento (il riferimento implicito `this`) di tipo `ClasseEstesa` (quindi al massimo del tipo della sottoclasse).

Al contrario non viene compilata la classe:

```

import prog.io.*;
import pack4.ClasseBase;

class ClasseEstesa extends ClasseBase {

    public ClasseEstesa(int x) {
        super(x);
    }

    public void usaProtected(ClasseBase c) {
        ConsoleOutputManager out = new ConsoleOutputManager();
        out.println("x = " + c.x);
        out.println(c.visibileAlleSottoclassi());
    }

    public static void main (String[] args) {
        ClasseEstesa e = new ClasseEstesa(3);
        e.usaProtected(new ClasseBase(3));
    }
}

```

In questo caso l'accesso ai membri protetti avviene attraverso una variabile riferimento di tipo `ClasseBase` (cioè il tipo della superclasse).

Per concludere la trattazione di `protected` osserviamo che un metodo `protected` di una superclasse può essere ridefinito `protected` o `public`.

8.14 Il modificatore `final`

Il modificatore `final` può essere applicato alle variabili, ai metodi e alle classi. Esso ha significati differenti a seconda del contesto in cui compare. In generale, comunque, stabilisce che l'identificatore cui è applicato non può essere modificato.

final applicato a variabili

Una variabile `final` (statica o no) è una variabile alla quale è possibile assegnare un valore una sola volta. Nel codice che la utilizza il valore della variabile è quindi costante. Un tentativo di modificare la variabile dopo che le è stato assegnato un valore dà luogo a un errore in fase di compilazione.

All'interno del codice di una classe, le variabili possono essere dichiarate in contesti diversi (campi, variabili locali e parametri dei metodi e dei costruttori sono dichiarazioni di variabile). Analizziamo le varie situazioni possibili.

Per quel che riguarda le variabili locali si può assegnare loro il valore in fase di dichiarazione o in un momento successivo. Ad esempio, nel caso delle istruzioni:

```
final int X;
final int Y = 2;
X = Y + 1;
```

a `Y` il valore viene assegnato al momento della dichiarazione, mentre a `X` viene assegnato in un momento successivo. Per le costanti, cioè per le variabili `final`, si utilizzano convenzionalmente nomi in maiuscolo, in modo da evidenziarne la peculiarità.

Nel caso delle variabili riferimento, poiché esse contengono il riferimento all'oggetto e non l'oggetto, ciò che risulta immodificabile è appunto il riferimento. Ad esempio viene compilato il frammento di codice:

```
final Rettangolo RETTANGOLO = new Rettangolo(2,3);
RETTANGOLO.cambiaBase(4);
```

Al contrario non viene compilato il frammento:

```
final String STRINGA = "pippo";
STRINGA = STRINGA.toUpperCase();
```

in cui alla variabile `final` `STRINGA` si assegna un riferimento a una stringa differente, quella restituita dall'espressione `STRINGA.toUpperCase()` dopo che un riferimento le era già stato assegnato in fase di definizione.

È possibile definire `final` il parametro di un metodo, come ad esempio in:

```
public void f(final int X) {
    ...
}
```

In questo caso, dato che alla variabile `X` viene assegnato un valore al momento dell'invocazione del metodo, all'interno del codice del metodo `f` essa risulterà immodificabile.

Nel caso dei campi statici il valore può essere assegnato solo in fase di definizione, in quanto, come sappiamo, ai campi statici viene assegnato un valore all'atto della creazione anche in mancanza di un'inizializzazione esplicita.

Nel caso dei campi non statici è possibile assegnargli un valore in fase di definizione, oppure all'interno dei costruttori. I campi `final` privi di un'inizializzazione esplicita sono chiamati, nella terminologia Java, *blank-final*. Consideriamo ad esempio la seguente classe A:

```

public class A {
    final int C = 0;
    final int D;

    public A() {
        D = 1;
    }

    public A(int x) {
        D = x;
    }

    public String toString() {
        return C + ", " + D;
    }
}

```

La variabile C è inizializzata esplicitamente a zero, mentre D è un blank-final e deve obbligatoriamente essere inizializzato all'interno di tutti i costruttori della classe. In caso contrario il compilatore segnalerà un errore.

final applicato a metodi

Applicato a un metodo, **final** stabilisce che il metodo non può essere ridefinito dalle estensioni della classe. La ragione principale per dichiarare **final** un metodo è quella di impedire alle sottoclassi di cambiarne il significato.⁹ Consideriamo ad esempio una classe **ParolaSegreta** i cui oggetti nascondano al proprio interno una parola inaccessibile fornendo un metodo **public boolean verifica(String altra)**, che restituisce **true** se e solo se la stringa fornita come argomento è uguale a quella memorizzata nell'oggetto. Analizziamo ora il seguente codice, in cui **in** e **out** sono i soliti collegamenti ai canali di comunicazione con la tastiera e con il video, e **p** è un riferimento a un oggetto di tipo **ParolaSegreta**.

```

boolean indovinato = false;
do {
    String tentativo = in.readLine("Inserisci una parola> ");
    if (p.verifica(tentativo))
        indovinato = true;
    else
        out.println("Ritenta sarai più fortunato.");
} while (!indovinato);

```

⁹ In realtà i metodi **final** assicurano anche una maggiore efficienza, perché la garanzia che non possono essere ridefiniti consente al compilatore di trattarli in modo diverso dagli altri metodi. Per maggiori dettagli si consulti il manuale del linguaggio.

Se il metodo `verifica` non è dichiarato `final`, possiamo estendere la sottoclasse ridefinendo il metodo con:

```
public class Imbroglio extends ParolaSegreta {
    public boolean verifica(String altra) {
        return true;
    }
}
```

In questo modo, se prima del codice precedente inseriamo l'assegnamento

```
p = new Imbroglio();
```

in virtù del polimorfismo, il metodo `verifica` eseguito nella condizione dell'`if` sarà quello di `Imbroglio`, che restituisce sempre `true`, e non quello della classe `ParolaSegreta`. Se il metodo `verifica` di `ParolaSegreta` fosse invece dichiarato `final`, non potremmo ridefinire il metodo `verifica` nella classe `Imbroglio` modificandone il contratto.

final applicato alle classi

Per quel che riguarda le classi, `final` specifica semplicemente che non possono essere estese. Il tentativo di estendere una classe `final` dà luogo a un errore in fase di compilazione. I motivi per definire una classe `final` sono inerenti a questioni di correttezza ed efficienza. Come nel caso dei metodi, ad esempio, si può voler impedire l'estensione di una classe in modo da violarne il contratto. Esempi di classi `final` di Java sono la classe `String` e le classi involucro.

Esercizi

8.11 Considerate le seguenti classi (in cui nella classe A sia importata la classe `ConsoleOutputManager`):

```
public class C {
    static int h = 3;
    private int x = 6;
    public int i = 7;

    public void inc() {
        x++;
        i++;
        h++;
    }

    public int sum() {
        return x + h + i;
```

```

    }
}

public class B extends C {
    public int i = 4, k = 5;

    public int sum() {
        int j = super.sum();
        j = j + i + k;
        return j;
    }
}

public class A {
    public static void main(String[] args) {
        //predisposizione del canale di comunicazione
        ConsoleOutputManager out = new ConsoleOutputManager();

        C vu = new C();
        out.println(vu.sum());
        vu = new B();
        out.println(vu.sum());
    }
}

```

(1) Per ognuna delle seguenti affermazioni indicate se è vera oppure falsa:

- (a) C ha un metodo statico
- (b) A è una superclasse di C e di B
- (c) ogni istanza di C ha un campo statico
- (d) B è una superclasse di C
- (e) C ha un costruttore
- (f) C ha un campo statico
- (g) C è una superclasse di B
- (h) nel codice di A si può utilizzare il campo k di B
- (i) A è una superclasse di C
- (j) nel codice di B si può utilizzare il campo x di C.

(2) Indicate l'output prodotto dal metodo main di A.

8.12 Considerate le seguenti classi:

```
public class A {
```

```

private int x;

public A(int z) {
    x = z;
}

public String toString() {
    return "x = " + x;
}

public String toString(String s) {
    return s + this.toString();
}

public class B extends A {
    private int y, w;

    public B(int a, int b) {
        y = a;
        w = b;
    }

    public B(int a, int b, int c) {
        super(a);
        y = b;
        w = c;
    }

    public String toString() {
        return super.toString() + ", w = " + w + ", y = " + y;
    }
}

```

- (1) Uno dei costruttori della classe B provoca un errore in compilazione: individuatene la causa e riscrivete la classe B eliminando tale costruttore.
- (2) Riscrivete il metodo `equals` di `Object` sia nella classe A che nella classe B, in modo da verificare l'uguaglianza di tutti i campi degli oggetti da confrontare (si noti che si chiede di "riscrivere" il metodo `equals`, non di definirne uno nuovo. In sostanza, il metodo che viene richiesto deve ricevere un parametro di tipo `Object`).
- (3) Considerate ora le seguenti variabili:

A alfa;

```
B beta;
```

Per ognuna delle seguenti chiamate di metodo, indicate se la chiamata è corretta e, in caso affermativo, la segnatura che sarà selezionata dal compilatore:

- (1) alfa.toString();
 - (2) alfa.toString(10);
 - (3) alfa.toString("10");
 - (4) beta.toString();
 - (5) beta.toString(10);
 - (6) beta.toString("10");
 - (7) alfa.toString(beta);
 - (8) beta.toString(alfa);
- (4) Considerate ora la seguente classe:

```
import prog.io.*;  
  
class Prova {  
    public static void main(String[] args) {  
        //predisposizione del canale di comunicazione  
        ConsoleOutputManager out = new ConsoleOutputManager();  
  
        A alfa;  
        B beta;  
        alfa = new A(4);  
        out.println(alfa.toString("risultato: "));  
        alfa = beta = new B(1, 2, 3);  
        out.println(alfa.toString("risultato: "));  
    }  
}
```

Indicate l'output prodotto dall'esecuzione del metodo `main` (prestate particolare attenzione al significato del riferimento `this`). Verificate sulla vostra macchina la risposta ottenuta.

- 8.13 Definite la classe `Ora` introdotta nell'Esercizio 7.5 estendendo la classe `Orario`. Osservate la differenza fra questa implementazione di `Ora` e quella sviluppata per l'Esercizio 7.5.

Tipi enumerativi, tipi generici e interfacce

Nella prima parte del testo abbiamo utilizzato i tipi enumerativi, i tipi generici e le interfacce. Nei capitoli precedenti abbiamo inoltre mostrato come scrivere classi che implementano una data interfaccia. In questo capitolo approfondiamo lo studio di questi argomenti, mostrando come possano essere definiti i tipi enumerativi, i tipi generici e le interfacce. Approfondiremo, inoltre, alcuni aspetti relativi a questi argomenti che, nei capitoli precedenti, sono stati solo accennati.

9.1 Definizione di tipi enumerativi

Come abbiamo accennato nel Paragrafo 4.7, i tipi enumerativi sono definiti tramite particolari classi. La loro dichiarazione si apre con la parola riservata `enum` e utilizza le stesse regole delle classi relativamente al nome del file e dei modificatori. All'interno del corpo del tipo enumerativo vanno elencati, prima di tutto, gli identificatori delle costanti del tipo enumerativo, separati da una “virgola” (,); la dichiarazione è conclusa, come tutte le istruzioni, dal carattere “punto e virgola”. Per esemplificare le caratteristiche dell'implementazione dei tipi enumerativi presentiamo l'esempio del tipo enumerativo `MeseDellAnno` i cui valori rappresentano appunto i mesi dell'anno.

```
public enum MeseDellAnno {  
    // COSTANTI ENUMERATIVE  
    GENNAIO, FEBBRAIO, MARZO, APRILE, MAGGIO, GIUGNO, LUGLIO, AGOSTO,  
    SETTEMBRE, OTTOBRE, NOVEMBRE, DICEMBRE;  
}
```

In fase di esecuzione, la Java Virtual Machine crea un oggetto per ogni identificatore indicato e ne memorizza il riferimento nella costante corrispondente. Ogni tipo enumerativo è una classe

che estende una particolare classe generica `Enum` definita nel package `java.lang`.¹ In questo modo i metodi definiti nella classe `Enum`, e ovviamente quelli definiti in `Object`, sono automaticamente disponibili per tutti gli oggetti del tipo enumerativo. I metodi `name`, `ordinal` e `toString` descritti nel Paragrafo 4.7 sono disponibili per ogni oggetto di tipo enumerativo in quanto ereditati dalla classe `Enum`. Inoltre tale classe implementa l’interfaccia `Comparable<E>` e fornisce la seguente implementazione del metodo `compareTo`:

- `public int compareTo(E o)`

Confronta l’oggetto di tipo enumerativo che esegue il metodo con quello fornito come argomento. Restituisce un valore negativo, zero o un valore positivo a seconda che l’oggetto che esegue il metodo preceda, sia uguale o segua quello fornito come argomento, sulla base dell’ordine in cui sono dichiarate le costanti nel tipo enumerativo.²

Ad esempio, nell’ordine implementato dal metodo `compareTo` di `MeseDellAnno`, GENNAIO è minore di FEBBRAIO che è minore di MARZO e così via.

Come osservato nel Paragrafo 5.5, un’altra caratteristica dei tipi enumerativi è quella di fornire un metodo statico `values` che restituisce un array contenente le costanti del tipo enumerativo nell’ordine in cui sono dichiarate. Ad esempio, il tipo enumerativo `MeseDellAnno` dispone automaticamente del metodo statico

```
public static MeseDellAnno[] values()
```

L’analogia fra classi e tipi enumerativi appare ancora più evidente nel momento in cui si osserva che, come per le classi, è possibile aggiungere ad un tipo enumerativo nuovi metodi o sovrascrivere i metodi esistenti.

Per prima cosa ridefiniamo il metodo `toString` del tipo `MeseDellAnno`. L’implementazione ereditata dalla classe `Enum` restituisce la stringa che rappresenta la costante del tipo enumerativo che esegue il metodo, cioè fornisce lo stesso risultato del metodo `name`. A tale scopo possiamo aggiungere al tipo enumerativo la seguente implementazione del metodo `toString`:

```
public String toString() {
    switch (this) {
        case GENNAIO:
            return "Gennaio";
        case FEBBRAIO:
            return "Febbraio";
        case MARZO:
            return "Marzo";
        case APRILE:
            return "Aprile";
        case MAGGIO:
            return "Maggio";
    }
}
```

¹ Senza entrare nei dettagli, ci limitiamo a segnalare che `Enum` ha un tipo parametro `E` vincolato a estendere `Enum<E>`, cioè la classe viene indicata con `Enum<E extends Enum<E>>`. Il tipo argomento utilizzato nell’invocazione di `Enum` è proprio il tipo enumerativo. Ad esempio `MeseDellAnno` estende `Enum<MeseDellAnno>`.

² Nel caso dei tipi enumerativi, il metodo `compareTo` è `final` e dunque non può essere sovrascritto.

```

        return "Maggio";
    case GIUGNO:
        return "Giugno";
    case LUGLIO:
        return "Luglio";
    case AGOSTO:
        return "Agosto";
    case SETTEMBRE:
        return "Settembre";
    case OTTOBRE:
        return "Ottobre";
    case NOVEMBRE:
        return "Novembre";
    case DICEMBRE:
        return "Dicembre";
    default:
        return "";
}
}

```

In questo modo il metodo `toString` ridefinito è disponibile per ogni oggetto del tipo enumerativo. Al momento dell'invocazione del metodo, il caso selezionato nell'istruzione `switch` sarà quello corrispondente all'istanza del tipo che esegue il metodo.³

Per definire nuovi comportamenti non dobbiamo far altro che implementare nuovi metodi nel corpo del tipo enumerativo. Ad esempio possiamo aggiungere a `MeseDellAnno` i metodi:

- `public MeseDellAnno precedente()`
Restituisce il riferimento all'oggetto che rappresenta il mese precedente a quello che esegue il metodo. L'oggetto precedente a `GENNAIO` è quello corrispondente a `DICEMBRE`.
- `public MeseDellAnno successivo()`
Restituisce il riferimento all'oggetto che rappresenta il mese successivo a quello che esegue il metodo. L'oggetto successivo a `DICEMBRE` è quello corrispondente a `GENNAIO`.

Una possibile implementazione del metodo `successivo` è la seguente:

```

public MeseDellAnno successivo() {
    return MeseDellAnno.values()[(this.ordinal() + 1) % 12];
}

```

Supponiamo ora di voler aggiungere un metodo di nome `numeroGiorni` che restituisca il numero dei giorni del mese che esegue il metodo (per ora non teniamo conto del fatto che l'anno può essere bisestile). Potremmo procedere in modo simile a quanto fatto per il metodo `toString` e

³ Il caso `default` va definito, anche se non verrà mai eseguito, per consentire la compilazione del metodo.

stabilire il valore da restituire in base all’oggetto che esegue il metodo. È possibile procedere in modo diverso aggiungendo comportamenti alle singole costanti definiti mediante metodi detti *metodi propri delle costanti*.⁴ Tali metodi devono essere implementati nel corpo delle costanti, delimitato, come al solito, da una coppia di parentesi graffe. Ad esempio, la struttura del tipo `MeseDellAnno` con l’implementazione di `numeroGiorni` come metodo proprio delle costanti è la seguente:

```
public enum MeseDellAnno {
    // COSTANTI ENUMERATIVE
    GENNAIO {
        public int numeroGiorni() {
            return 31;
        }
    },
    FEBBRAIO {
        public int numeroGiorni() {
            return 28;
        }
    },
    MARZO {
        public int numeroGiorni() {
            return 31;
        }
    },
    ...
    public abstract int numeroGiorni();
    ...
}
```

Si osservi che il metodo `numeroGiorni` è stato dichiarato astratto nel corpo del tipo enumerativo. Senza tale dichiarazione il metodo non può essere invocato: se cioè `m` fosse una variabile riferimento di tipo `MeseDellAnno`, non verrebbe compilata l’invocazione

```
m.numeroGiorni()
```

La dichiarazione del metodo astratto nel corpo del tipo enumerativo è simile a quella data nel corpo delle classi astratte: essa costituisce una promessa di implementazione del metodo nel corpo delle costanti. Ovviamente se tale promessa non viene mantenuta, cioè se il metodo non viene definito nel corpo di ogni costante, il compilatore segnala un errore.

Presentiamo ora un’implementazione più elegante della classe `MeseDellAnno` che sfrutta la possibilità di associare dei campi agli oggetti del tipo enumerativo. In particolare, associamo a ogni mese un campo di tipo `String` per memorizzare il nome del mese e uno di tipo `int` per

⁴ *Constant-specific methods*.

memorizzare il numero dei giorni del mese. La dichiarazione dei campi in un tipo enumerativo segue le stesse regole date per le classi.

Come nel caso delle classi, per inizializzare in modo opportuno i campi degli oggetti dobbiamo definire un costruttore. I costruttori di un tipo enumerativo tuttavia hanno un ruolo particolare in quanto vengono utilizzati esclusivamente dalla Java Virtual Machine al momento della costruzione degli oggetti del tipo. Per questa ragione non possono essere dichiarati `public`. Alla definizione di ciascuna costante del tipo enumerativo viene fatta seguire, fra parentesi tonde, la lista degli argomenti per il costruttore. Ad esempio la classe `MeseDellAnno` può essere definita con la seguente struttura:

```
public enum MeseDellAnno {
    // COSTANTI ENUMERATIVE
    GENNAIO("Gennaio", 31), FEBBRAIO("Febbraio", 28), MARZO("Marzo", 31),
    APRILE("Aprile", 30), MAGGIO("Maggio", 31), GIUGNO("Giugno", 30),
    LUGLIO("Luglio", 31), AGOSTO("Agosto", 31), SETTEMBRE("Settembre", 30),
    OTTOBRE("Ottobre", 31), NOVEMBRE("Novembre", 30),
    DICEMBRE("Dicembre", 31);

    // CAMPI
    private String nome;
    private int numGiorni;

    // COSTRUTTORI
    private MeseDellAnno(String nome, int numGiorni) {
        this.nome = nome;
        this.numGiorni = numGiorni;
    }
    ...
}
```

In fase di esecuzione la Java Virtual Machine, per costruire ciascuno degli oggetti corrispondenti alle costanti del tipo enumerativo, invoca il costruttore istanziando i parametri formali con gli argomenti indicati dopo il nome della costante. Pertanto, in questo modo, per ognuna delle costanti verrà fabbricato un oggetto con i campi `nome` e `numGiorni` inizializzati con i valori indicati dopo la costante.

In questo caso vi è un unico costruttore, ma in generale può esservene più di uno: la Java Virtual Machine seleziona quello da eseguire in base alla segnatura nel modo usuale.

A questo punto possiamo implementare i metodi `toString` e `numeroGiorni` nel modo seguente:

```
public int numeroGiorni() {
    return this.numGiorni;
}
```

```
public String toString() {
    return this.nome;
}
```

Per concludere mostriamo in che modo si possono implementare i metodi:

- `public int numeroGiorni(int anno)`
Restituisce il numero dei giorni del mese, relativamente all'anno specificato come argomento. Pertanto, nel caso degli anni bisestili, per il mese di febbraio restituisce 29.
- `public int numeroGiorni(boolean bisestile)`
Restituisce il numero dei giorni del mese; nel caso di febbraio se l'argomento è `true` restituisce 29, se è `false` restituisce 28.

Sviluppiamo prima di tutto quest'ultimo metodo. Il suo comportamento differisce da quello del metodo `numeroGiorni`, già implementato, solo nel caso che a eseguirlo sia l'oggetto a cui fa riferimento `FEBBRAIO`. Procediamo dunque nel modo seguente: prima di tutto implementiamo il metodo `numeroGiorni(boolean bisestile)` nel tipo `MeseDellAnno` richiamando il metodo `numeroGiorni()` in modo che esso sia disponibile per ogni costante.

```
public int numeroGiorni(boolean bisestile) {
    return this.numeroGiorni();
}
```

Quindi riscriviamo `numeroGiorni(boolean bisestile)` come metodo proprio della costante `FEBBRAIO` nel modo seguente:

```
public int numeroGiorni(boolean bisestile) {
    return bisestile ? 29 : 28;
}
```

Si osservi che questa soluzione è un'elegante e utile combinazione di overloading e overriding.

Implementiamo infine il metodo `numeroGiorni` che riceve come parametro l'anno. Tale metodo invoca quello appena implementato, fornendo come argomento `true` o `false`, a seconda che l'anno considerato sia bisestile o no. Per controllare se un anno è bisestile possiamo utilizzare il metodo `isInAnnoBisextile` della classe `Data`: in particolare costruiamo una qualunque data dell'anno considerato alla quale chiediamo di eseguire tale metodo:

```
public int numeroGiorni(int anno) {
    Data d = new Data(1, 1, anno);
    return this.numeroGiorni(d.isInAnnoBisextile());
}
```

Per riassumere riportiamo il codice completo della classe `MeseDellAnno` così sviluppata.

```
import prog.utili.Data;

public enum MeseDellAnno {
    // COSTANTI ENUMERATIVE
    GENNAIO("Gennaio", 31),
    FEBBRAIO("Febbraio", 28) {
        public int numeroGiorni(boolean bisestile) {
            return bisestile ? 29 : 28;
        }
    },
    MARZO("Marzo", 31), APRILE("Aprile", 30), MAGGIO("Maggio", 31),
    GIUGNO("Giugno", 30), LUGLIO("Luglio", 31), AGOSTO("Agosto", 31),
    SETTEMBRE("Settembre", 30), OTTOBRE("Ottobre", 31),
    NOVEMBRE("Novembre", 30), DICEMBRE("Dicembre", 31);

    // CAMPI
    private String nome;
    private int numGiorni;

    // COSTRUTTORI
    private MeseDellAnno(String nome, int numGiorni) {
        this.nome = nome;
        this.numGiorni = numGiorni;
    }

    // METODI
    public int numeroGiorni() {
        return this.numGiorni;
    }

    public int numeroGiorni(boolean bisestile) {
        return this.numeroGiorni();
    }

    public int numeroGiorni(int anno) {
        Data d = new Data(1, 1, anno);
        return this.numeroGiorni(d.isInAnnoBisextile());
    }

    public MeseDellAnno successivo() {
        return MeseDellAnno.values()[(this.ordinal() + 1) % 12];
    }
}
```

```

    }

    public MeseDellAnno precedente() {
        return MeseDellAnno.values()[this.ordinal() - 1 < 0 ? 11
            : this.ordinal() - 1];
    }
}

```

Per chiarire le idee sul meccanismo utilizzato per la definizione di tipi enumerativi possiamo immaginarci che il tipo enumerativo definisca una classe (nel nostro caso `MeseDellAnno`) il cui stato è costituito dai campi definiti nel corpo del tipo enumerativo e il cui comportamento è costituito dai metodi ereditati dalla classe `Enum` (e ovviamente da `Object`) e dai metodi definiti nel corpo del tipo enumerativo. Ogni costante del tipo enumerativo fa riferimento a un oggetto costruito a partire da una sottoclasse di quella corrispondente al tipo enumerativo. Questa sottoclasse eredita i metodi di quella corrispondente al tipo enumerativo e può definirne di propri (quelli dichiarati nel corpo della costante); se nel corpo della costante è dichiarato un metodo con segnatura uguale a quella di un metodo della classe corrispondente al tipo enumerativo (come il caso del metodo `numGiorni(boolean bisestile)` per la costante `FEBBRAIO`) il metodo viene sovrascritto.⁵

Infine va osservato che, come per le classi, è possibile definire in un tipo enumerativo campi e metodi statici. Ad esempio potremmo aggiungere a `MeseDellAnno` il seguente metodo statico che preso come argomento l'intero fra 1 e 12 che rappresenta un mese restituisce il corrispondente valore enumerativo:

```

public static MeseDellAnno getMese(int m) {
    return values()[m - 1];
}

```

Si osservi che l'esecuzione provoca un errore se il valore specificato come argomento non è compreso fra 1 e 12.

Esercizi

- 9.1 Progettate e implementate un tipo enumerativo `Stagione`, le cui costanti rappresentino le quattro stagioni dell'anno. Il tipo dovrà fornire i metodi `meseInizio` e `meseFine` che restituiscono il mese iniziale e il mese finale di una stagione. Inoltre, si dovranno fornire dei metodi `dataInizio` e `dataFine` che, privi di argomenti, restituiscano rispettivamente la data di inizio e la data di fine della stagione rispetto all'anno corrente. Ad esempio, se la data corrente è 2 aprile 2005, i metodi per la costante corrispondente all'inverno dovranno restituire rispettivamente le date 21 dicembre 2005 e 20 marzo 2006. Realizzate due ulteriori versioni di questi metodi. La prima che riceva come parametro una data, rispetto alla quale fornire il risultato, la seconda che riceva un anno (utilizzate le classi

⁵ Nel corpo di una costante enumerativa è possibile dichiarare anche dei campi.

`MeseDellAnno` e `Data`). Aggiungete anche un metodo statico che, ricevendo una data, restituisca la stagione a cui appartiene. Scrivete poi un'applicazione di prova per la classe `Stagione`.

- 9.2 Utilizzando la classe `Stagione` scrivete un'applicazione che legga il numero di un anno e stampi un calendario, per l'anno, suddiviso in mesi. All'inizio di ciascun mese si dovrà indicare a quale stagione appartiene, evidenziando gli eventuali cambi di stagione.
- 9.3 Realizzate un tipo enumerativo `Seme` le cui costanti rappresentino i quattro semi delle carte da gioco (cuori, quadri, fiori e picche). Realizzate poi un secondo tipo enumerativo `ValoriCarte` le cui costanti rappresentino i valori che compaiono sulle carte di un mazzo da 52 (da due a dieci, fante, donna, re, asso). Può essere utile fornire un metodo che associa a ciascuna carta un punteggio. Definite poi una classe `Carta` che rappresenti una carta da gioco, e una classe `Mazzo` che definisca un mazzo di 52 carte. La classe `Carta` avrà un costruttore che riceve come argomenti il valore e il seme della carta da rappresentare. La classe `Mazzo` avrà un costruttore privo di argomenti che costruisce un nuovo mazzo di carte. In particolare la classe `Mazzo` può essere implementata utilizzando una Sequenza di oggetti di tipo `Carta`. Il costruttore può essere implementato costruendo una sequenza vuota e aggiungendovi poi tutte le possibili carte (che possono essere costruite mediante due cicli `for-each` innestati: in uno l'indice varia sui semi, nell'altro sui valori). Nella classe `Mazzo` fornite anche un metodo `mescola` per mischiare le carte (la classe `Sequenza` dispone a sua volta di un metodo `mescola` che mischia gli elementi della sequenza) e un metodo `toString` che restituisca il contenuto dell'intero mazzo. Scrivete poi una classe di prova per ciò che avete implementato.
- 9.4 Progettate ulteriori metodi per la classe `Mazzo` e per le altre classi dell'Esercizio 9.3, che permettano di simulare alcuni giochi con le carte.

9.2 Definizione di classi generiche

In questo paragrafo, al fine di illustrare come possano essere definite le classi generiche, sviluppiamo una classe `Coppia` con due tipi parametri E e F. Le istanze di `Coppia` sono coppie di oggetti, il primo di tipo E ed il secondo di tipo F. Ad esempio, un'indicazione di tempo può essere modellata come una coppia formata da un oggetto di tipo `Data` e da un oggetto di tipo `Orario`, cioè da un'istanza di `Coppia<Data, Orario>`.

La classe avrà un costruttore con due argomenti, rispettivamente dei tipi argomenti corrispondenti ai tipi parametri E ed F. Tale costruttore fabbrica un oggetto che rappresenta la coppia degli oggetti forniti. Si consideri ad esempio il seguente frammento di codice:

```
 Data d = new Data(22, 1, 2005);
 Orario o = new Orario(23, 33);
 Coppia<Data, Orario> tempo = new Coppia<Data, Orario>(d, o);
 Coppia<Data, Orario> adesso =
    new Coppia<Data, Orario>(new Data(), new Orario());
```

L'oggetto riferito dalla variabile `tempo` rappresenta le ore 23.33 del giorno 22 gennaio 2005, l'oggetto riferito dalla variabile `adesso` rappresenta l'istante (data del calendario e orario) in cui viene eseguito il codice.

Nella classe `Coppia` forniamo alcuni semplici metodi:

- `public E getSinistro()`
Restituisce il primo elemento della coppia, cioè quello di sinistra.
- `public F getDestro()`
Restituisce il secondo elemento della coppia, cioè quello di destra.
- `public String toString()`
Restituisce una stringa contenente le stringhe associate ai due oggetti che costituiscono la coppia.
- `public static int numeroCoppie()`
Restituisce il numero totale di istanze della classe costruite.

La classe viene implementata con due campi privati, `sinistro` di tipo `E` e `destro` di tipo `F`, che memorizzano i riferimenti ai due oggetti che costituiscono la coppia. Vi è inoltre un campo statico `nCoppie` di tipo `int`, inizializzato a zero e incrementato a ogni invocazione del costruttore, che tiene traccia del numero totale di coppie costruite. Tale campo è utile per implementare il metodo `numeroCoppie`.

Nell'intestazione della classe vengono indicati, dopo il nome, gli identificatori dei tipi parametri formali, separati da una "virgola" (,). Questi identificatori possono essere scelti liberamente dal programmatore e vengono utilizzati nel codice della classe. Per questioni di leggibilità, è convenzione utilizzare identificatori costituiti da un'unica lettera maiuscola, in genere `E`, `F`, ... o `S`, `T`

Ecco l'implementazione della classe `Coppia`:

```
public class Coppia<E, F> {
    //CAMPI
    private E sinistro;
    private F destro;
    private static int nCoppie = 0;

    //COSTRUTTORI
    public Coppia(E e, F f) {
        sinistro = e;
        destro = f;
        nCoppie++;
    }

    //METODI
    public E getSinistro() {
```

```

    return sinistro;
}

public F getDestro() {
    return destro;
}

public String toString() {
    return "(" + sinistro + ", " + destro + ")";
}

//METODI STATICI
public static int numeroCoppie() {
    return nCoppie;
}
}

```

Sottolineiamo quanto già indicato nel Paragrafo 6.18: una classe generica non viene duplicata per ogni suo tipo parametrizzato, ma rimane unica. In uno specifico tipo parametrizzato, come `Coppia<Data, Orario>`, i tipi argomento vengono sostituiti ai tipi parametro, senza però creare una nuova classe. Di conseguenza, le parti statiche della classe, non essendo legate a una specifica invocazione della classe stessa, *non possono fare riferimento ai tipi parametro*.

Possiamo osservare, ad esempio, che il campo statico `nCoppie` della classe `Coppia` è unico. Consideriamo la seguente classe di prova, in cui la classe `Coppia` viene utilizzata prima con tipi argomento `Data` e `Orario`, costruendo due oggetti, e poi con tipi argomento `String` e `Integer`, costruendo un solo oggetto. Alla fine dell'esecuzione, il metodo `numeroCoppie`, che restituisce il contenuto del campo `nCoppie`, fornisce come risultato 3, cioè il numero complessivo di coppie costruite, indipendentemente dai tipi argomento:

```

import prog.io.*;
import prog.utili.Data;
import prog.utili.Orario;

class ProvaCoppia {

    public static void main(String[] a) {
        ConsoleOutputManager out = new ConsoleOutputManager();
        out.println("Numero coppie: " + Coppia.numeroCoppie());

        Data d = new Data(22, 1, 2005);
        Orario o = new Orario(23, 33);
        Coppia<Data, Orario> tempo = new Coppia<Data, Orario>(d, o);
        Coppia<Data, Orario> adesso =

```

```

        new Coppia<Data, Orario>(new Data(), new Orario());
        out.println("Tempo: " + tempo);
        out.println("Adesso: " + adesso);
        out.println("Numero coppie: " + Coppia.numeroCoppie());

Coppia<String, Integer> x = new Coppia<String, Integer>("Pippo", 4);
out.println("x: " + x);
out.println("Numero coppie: " + Coppia.numeroCoppie());
}
}

```

Segue il risultato di un'esecuzione dell'applicazione ProvaCoppia:

```

Numero coppie: 0
Tempo: (22.01.2005, 23:33)
Adesso: (23.01.2005, 0:17)
Numero coppie: 2
x: (Pippo, 4)
Numero coppie: 3

```

In questo paragrafo, al fine di concentrare l'attenzione sulla scrittura di un tipo generico, abbiamo mostrato un esempio estremamente semplice. Molto spesso i tipi generici sono utilizzati per rappresentare collezioni di dati. Approfondiremo questi aspetti nel Capitolo 12.

Esercizi

9.5 Progettate una classe **Istante**, i cui oggetti rappresentino istanti di tempo. La classe dovrebbe fornire dei metodi per ricavare la data, l'orario o anche i singoli elementi (come mese, giorno, ora, etc.), e dei metodi per confrontare due istanti e per calcolarne la differenza, in termini di minuti, di giorni e di anni. Implementate poi la classe utilizzando, come unico campo, un'istanza di **Coppia<Data, Orario>**.

9.6 Scrivete un metodo **equals** da aggiungere alla classe **Coppia**.

9.7 Implementate una sottoclasse di **Coppia**, di nome **CoppiaMonotipo**, con un solo tipo argomento. Le istanze di **CoppiaMonotipo** sono coppie di oggetti del tipo argomento. Ad esempio, le istanze di **CoppiaMonotipo<String>** sono coppie di stringhe. La classe fornirà un costruttore con due argomenti e gli stessi metodi di **Coppia**.

9.8 Scrivete un'implementazione della classe generica **Occorrenza**, presentata nel Paragrafo 6.11.

9.3 Metodi generici

Oltre a tipi generici, è possibile anche definire *metodi generici*, cioè metodi dove uno o più tipi sono parametrizzati. Illustriamo questa possibilità con un semplice esempio.

Supponiamo di voler aggiungere alla classe `Coppia` un costruttore che riceva un oggetto e costruisca la coppia identica, cioè la coppia in cui il primo e il secondo elemento coincidono con l'oggetto ricevuto. Potremmo tentare di scrivere il costruttore in questo modo:

```
public Coppia(E e) {
    this(e, e);
}
```

Il costruttore non fa altro che delegare il proprio compito al costruttore con due argomenti, fornendo entrambi gli argomenti uguali al parametro ricevuto. Questa soluzione, apparentemente semplice, non è tuttavia praticabile: nella chiamata tramite `this` del costruttore con due argomenti, entrambi sono di tipo `E`, mentre il costruttore si aspetta il secondo argomento di tipo `F`. Per questa ragione, il compilatore fornisce un'indicazione d'errore sull'invocazione di `this`.⁶

Per evitare questo problema, al posto di fornire un costruttore, scriviamo un metodo statico da porre nella classe `Coppia` che svolga la stessa funzione. Cominciamo con lo scrivere un metodo che riceve un riferimento a un oggetto generico e crea la coppia identica a partire da questo oggetto:

```
public static Coppia creaIdentica(Object o) {
    return new Coppia(o, o);
}
```

Con questa scrittura, il compilatore fornirà degli avvertimenti relativi a operazioni che potrebbero provocare problemi in esecuzione. Infatti, non abbiamo specificato i tipi argomento di `Coppia`. Potremmo riscrivere il metodo indicando come argomenti `Object`:

```
public static Coppia<Object, Object> creaIdentica(Object o) {
    return new Coppia<Object, Object>(o, o);
}
```

In questo modo il compilatore non indicherà alcun problema. Tuttavia, così facendo, non abbiamo raggiunto l'obiettivo prefissato. Ad esempio, invocando:

```
Coppia.creaIdentica("pippo");
```

⁶ Si osservi che utilizzando `this` in un costruttore per richiamare un altro costruttore della stessa classe, come in questo caso, *non deve essere indicato* il tipo argomento. Ad esempio, con la scrittura `this<E, E>(e, e)` il compilatore segnalerebbe un errore immediatamente dopo la parola `this`. Infatti, qui non vi è alcuna necessità di specificare i tipi argomento: essi verranno già indicati nel codice che richiama il primo costruttore. Analogamente, nel caso dell'invocazione del costruttore della superclasse mediante `super`, i tipi argomento non devono essere indicati: eventuali relazioni tra i tipi argomento della sottoclasse e quelli della superclasse sono indicate direttamente nell'intestazione della sottoclasse stessa. Ad esempio, per la classe `CoppiaMonotipo` dell'Esercizio 9.7, la relazione può essere specificata scrivendo nell'intestazione `class CoppiaMonotipo<E> extends Coppia<E, E>`.

costruiremo una coppia in cui i due elementi coincidono con la stringa "pippo". Tuttavia, tale coppia, e il riferimento restituito, saranno di tipo `Coppia<Object, Object>` e non di tipo `Coppia<String, String>`, come desiderato. Potremmo risolvere il problema, per il tipo `String`, mediante l'overloading, aggiungendo un metodo specifico:

```
public static Coppia<String, String> creaIdentica(String o) {
    return new Coppia<String, String>(o, o);
}
```

tuttavia il problema si pone per ogni altro tipo riferimento.

In generale, per ogni tipo riferimento `T`, dovremmo aggiungere alla classe il seguente metodo:

```
public static Coppia<T, T> creaIdentica(T t) {
    return new Coppia<T, T>(t, t);
}
```

Affinché il metodo sia utilizzabile *per ogni* tipo riferimento `T`, e non solo per un `T` specifico, possiamo definirlo come metodo generico, indicando che `T` è un tipo parametro. I metodi generici vengono scritti, come i metodi usuali, elencando i tipi parametro (con una notazione identica ai tipi parametro delle classi) immediatamente prima del tipo restituito. Pertanto, il metodo verrà scritto come:

```
public static <T> Coppia<T, T> creaIdentica(T t) {
    return new Coppia<T, T>(t, t);
}
```

Il compilatore sostituirà al tipo parametro il tipo argomento corrispondente, stabilito analizzando la chiamata. Si considerino ad esempio le chiamate:

```
Coppia.creaIdentica("pippo")
Coppia.creaIdentica(10)
Coppia.creaIdentica(new Data())
```

La prima chiamata restituisce un riferimento di tipo `Coppia<String, String>`, la seconda di tipo `Coppia<Integer, Integer>`,⁷ la terza di tipo `Coppia<Data, Data>`.⁸

I metodi generici sono utilizzati solitamente quando vi sia un legame tra il tipo degli argomenti del metodo e il tipo del risultato, o quando sia necessario utilizzare, per implementare il metodo, il nome del tipo parametro. Come per i tipi argomento delle classi (si veda il Paragrafo 6.17), è possibile introdurre dei vincoli sui tipi parametro dei metodi, utilizzando le parole riservate `extends` e `super`. Il seguente metodo, ad esempio, riceve una sequenza di oggetti di un tipo `T` e il riferimento `t` a un oggetto di tipo `T`. Il metodo restituisce il numero di elementi della

⁷ Si noti in questo caso l'uso dell'autoboxing.

⁸ In questi esempi il tipo restituito dalle chiamate appare ovvio. È possibile tuttavia presentare esempi più complessi in cui ciò non accade. Il compilatore utilizza un algoritmo per inferire, sulla base dell'utilizzo del metodo, il tipo argomento adatto. I dettagli esulano però dallo scopo della nostra trattazione.

sequenza che risultano minori dell'oggetto riferito da `t`. In questo caso il tipo `T` non può essere un tipo qualunque, ma deve possedere un ordine totale. Come discusso nel Paragrafo 6.17, ciò è garantito se `T` è un sottotipo di `Comparable<? super T>`.

```
public static <T extends Comparable<? super T>>
    int contaMinoriDi(Sequenza<T> s, T t) {
    int n = 0;
    for (T o : s)
        if (o.compareTo(t) < 0)
            n++;
    return n;
}
```

Mostriamo infine il seguente esempio, nel quale l'utilizzo di un metodo generico è inutile:

```
public static <T> void stampaSequenza(Sequenza<T> s) {
    for (Object o: s)
        System.out.println(o);
}
```

In questa situazione il tipo `T` è irrilevante per il metodo: esso non è mai utilizzato all'interno del metodo, né per il tipo del risultato. In questo caso è consigliabile scrivere il metodo utilizzando, per il parametro, il supertipo `Sequenza<?>`:

```
public static void stampaSequenza(Sequenza<?> s) {
    for (Object o: s)
        System.out.println(o);
}
```

Esercizi

- 9.9 Scrivete un metodo generico statico, da aggiungere alla classe `Coppia`, che riceva come argomento il riferimento a un array di tipo base `T`, e restituisca il riferimento a una coppia di elementi di tipo `T`: se l'argomento fornito è `null`, il metodo dovrà restituire `null`; se è un array di zero elementi, il metodo dovrà restituire una coppia formata da due riferimenti `null`; se è un array di un solo elemento, il metodo dovrà restituire una coppia in cui il primo elemento è l'elemento dell'array e il secondo è `null`; se, infine, è un array di almeno due elementi, il metodo dovrà restituire la coppia formata dai primi due elementi dell'array.
- 9.10 Scrivete un metodo statico che riceva come parametro una sequenza di oggetti di un tipo che implementa `Comparable` e restituisca `true` se la sequenza è ordinata in modo crescente (ispirarsi al metodo `contaMinoriDi`).

- 9.11 Scrivete un metodo statico che riceva come parametro una sequenza di oggetti di un tipo che implementa Comparable e restituisca il minimo.
- 9.12 Scrivete un metodo statico che riceva come parametro una sequenza e un riferimento o di tipo Object e restituisca il numero di oggetti nella sequenza uguali a quello riferito da o, secondo il criterio di uguaglianza fornito dal metodo equals. In una prima versione scrivete il metodo come generico, con tipo parametro T e primo parametro formale di tipo Sequenza<T>. Riscrivete poi il metodo definendo il primo parametro formale di tipo Sequenza<?>.
- 9.13 La classe SequenzaOrdinata fornisce un metodo statico che, ricevendo come argomento un riferimento a una sequenza, restituisce il riferimento a una sequenza ordinata formata dagli stessi elementi della sequenza data. Naturalmente, la sequenza deve essere di tipo “ordinabile”. Il prototipo del metodo è il seguente:

```
public static <T extends Comparable<? super T>>
    SequenzaOrdinata<T> fromSequenza(Sequenza<T> s)
```

Scrivete il codice del metodo servendovi del costruttore privo di argomenti e del metodo add di SequenzaOrdinata (non è necessario conoscere l’implementazione della classe).

9.4 Definizione di interfacce

Oltre a utilizzare le interfacce disponibili nella distribuzione di Java, il programmatore può definirne di nuove. Lo schema dell’intestazione di un’interfaccia è simile a quello delle classi, a parte il fatto che la parola chiave utilizzata è `interface` anziché `class`. Inoltre all’interno del corpo dell’interfaccia è possibile specificare solo i prototipi dei metodi e non la loro implementazione.

A titolo d’esempio mostriamo la definizione dell’interfaccia generica Comparable definita nel package `java.lang`:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Si noti che nel prototipo del metodo non c’è il modificatore `public`: esso è sottinteso in quanto *tutti i metodi di un’interfaccia sono public*. È importante ricordare ciò perché, implementando un’interfaccia, il metodo definito nella classe deve avere esattamente lo stesso prototipo indicato in essa: implementando l’interfaccia Comparable in particolare è obbligatorio definire `public` il metodo `compareTo`.

Si può definire un’interfaccia estendendo interfacce già definite tramite la parola chiave `extends`. È essenziale osservare che, a differenza di quanto accade per le classi, è possibile far estendere più interfacce a un’interfaccia. Se I1 e I2 sono interfacce, possiamo ad esempio definire un’interfaccia I che le estende entrambe in questo modo:

```
public interface I extends I1, I2 {
    ...
}
```

Ovviamente un oggetto di una classe che implementa l'interfaccia I può essere visto sia come un oggetto di tipo I, sia come un oggetto di tipo I1, sia come un oggetto di tipo I2.

Le interfacce non hanno costruttori (in quanto non è possibile costruire istanze) e non possono avere campi; possono però contenere costanti statiche (cioè campi static e final).

9.5 Uso del supertipo definito dall'interfaccia

Vedremo ora come scrivere metodi di uso generale ricorrendo alle interfacce. In questo paragrafo definiremo in particolare alcuni metodi statici di una classe di utilità per gli array. Questa classe, che fa parte del package `prog.utili`, si chiama `GestioneArray`, e contiene metodi per ordinare array e per effettuare la ricerca di un elemento in un array. Prima di affrontare in generale il problema di ordinare oggetti di classi che implementano `Comparable`, introduciamo un algoritmo di ordinamento di un array di interi.

Ordinamento di un array di int

Il nostro obiettivo è quello di scrivere un metodo che permetta di ordinare un array di numeri interi. Dato che un tale array rappresenta una sequenza di numeri, iniziamo a vedere come si potrebbe procedere per ordinare in modo crescente una sequenza di 8 numeri interi. Consideriamo, ad esempio, la sequenza di numeri:

7 6 3 8 4 2 10 5

Per ordinare in maniera crescente una sequenza di numeri possiamo utilizzare una tecnica nota con il nome di *bubblesort*. Si inizia a scandire la sequenza di numeri da sinistra verso destra confrontando 7 e 6. Dato che questi numeri non sono ordinati, vengono scambiati, e la sequenza diventa:

6 7 3 8 4 2 10 5

Si prosegue quindi confrontando e scambiando 7 e 3, ottenendo così:

6 3 7 8 4 2 10 5

La coppia successiva, formata da 7 e 8 è ordinata, mentre 8 e 4 no. Pertanto si procede allo scambio della coppia 8 e 4 ottenendo:

6 3 7 4 8 2 10 5

Proseguendo la scansione fino alla fine della sequenza si ottiene:

6 3 7 4 2 8 5 10

Si passa quindi a una nuova scansione con la stessa tecnica, alla fine della quale si ha la sequenza:

3 6 4 2 7 5 8 10

Il procedimento viene iterato finché, avendo ottenuto la sequenza ordinata:

2 3 4 5 6 7 8 10

si riesce a effettuare un'intera scansione senza scambi.

Possiamo schematizzare l'algoritmo come:

```
do {
    effettua una scansione
} while (nella scansione vi sono stati scambi);
```

e, più in dettaglio, come:

```
do {
    scambiato = false;
    for ( ... )
        if (gli elementi di posto i - 1 e i non sono in ordine) {
            scambiali;
            scambiato = true;
        }
} while (scambiato);
```

Dato che vogliamo definire un metodo per l'ordinamento di generici array di interi, possiamo racchiudere il codice per l'ordinamento in un metodo statico (che porremo nella classe di utilità **GestioneArray**). In questo caso il metodo dovrà ricevere come argomento l'array da ordinare, quindi il prototipo sarà:

```
public static void ordina(int[] a)
```

Il ciclo **for** partirà dalla posizione 1 (che va confrontata con la posizione 0) e arriverà fino alla fine dell'array, cioè fino alla posizione **a.length**. Per scambiare due elementi dell'array dovremo inoltre utilizzare un'altra variabile: **temp**. Il codice completo del metodo è il seguente:

```
public static void ordina(int[] a) {
    int temp;
    boolean scambiato;
    do {
        scambiato = false;
        for (int i = 1; i < a.length; i++)
            if (a[i - 1] > a[i]) {
                temp = a[i - 1];
                a[i - 1] = a[i];
                a[i] = temp;
                scambiato = true;
            }
    } while (scambiato);
```

```

        scambiato = true;
    }
} while (scambiato);
}

```

Ordinamento di un array di String e di un array di Integer

Osserviamo che la sola proprietà del tipo `int` utilizzata dal metodo di ordinamento che abbiamo appena descritto è l'ordinamento totale definito sui suoi valori. Possiamo pensare quindi di generalizzarlo per definire metodi di ordinamento per altri tipi su cui sia definito un ordine totale. Possiamo ad esempio utilizzare il medesimo algoritmo per ordinare un array di stringhe. Nel caso delle stringhe, l'ordinamento totale è definito dal metodo

```
public int compareTo(String s)
```

della classe `String`. Tale metodo restituisce un valore negativo se la stringa che lo esegue precede, in ordine alfabetico, quella fornita tramite il parametro; restituisce zero nel caso le due stringhe coincidano; restituisce un valore positivo se la stringa che lo esegue segue quella fornita tramite il parametro. L'ordine alfabetico è determinato dalla tabella Unicode. Si osservi che questo metodo è quello definito dall'interfaccia `Comparable`. In particolare la classe `String` implementa il tipo parametrizzato `Comparable<String>`.

Utilizzando questo metodo `compareTo` possiamo riscrivere il precedente algoritmo di ordinamento per array di stringhe nel modo seguente:

```

public static void ordina(String[] a) {
    String temp;
    boolean scambiato;
    do {
        scambiato = false;
        for (int i = 1; i < a.length; i++)
            if (a[i - 1].compareTo(a[i]) > 0) {
                temp = a[i - 1];
                a[i - 1] = a[i];
                a[i] = temp;
                scambiato = true;
            }
    } while (scambiato);
}

```

Anche la classe involucro `Integer`, corrispondente al tipo primitivo `int`, implementa l'interfaccia `Comparable`. In particolare essa implementa `Comparable<Integer>` e fornisce un metodo di nome `compareTo` che permette di confrontare l'oggetto che esegue il metodo con un altro fornito tramite il parametro. Se l'oggetto che esegue il metodo rappresenta un intero maggiore di quello fornito tramite il parametro, il metodo restituisce un valore maggiore di zero; se, al

contrario, l’oggetto che esegue il metodo rappresenta un intero minore di quello fornito tramite il parametro, il metodo restituisce un risultato negativo; infine, se i due interi coincidono, il metodo restituisce zero. Quindi possiamo definire un metodo di ordinamento per array di Integer procedendo analogamente al caso di array di stringhe:

```
public static void ordina(Integer[] a) {
    Integer temp;
    boolean scambiato;
    do {
        scambiato = false;
        for (int i = 1; i < a.length; i++)
            if (a[i - 1].compareTo(a[i]) > 0) {
                temp = a[i - 1];
                a[i - 1] = a[i];
                a[i] = temp;
                scambiato = true;
            }
    } while (scambiato);
}
```

Ordinamento di un array di Comparable

Osserviamo che il codice dei due metodi precedenti è identico: essi ordinano un array basandosi sul solo presupposto che gli oggetti da ordinare mettano a disposizione il metodo `compareTo` previsto dall’interfaccia `Comparable`. Se volessimo scrivere un qualunque metodo di ordinamento per un array di oggetti di un tipo `T` che implementa `Comparable`, le uniche modifiche nel codice precedente sarebbero nella dichiarazione della variabile `temp`, di tipo `T`, e nella dichiarazione del parametro `a` del metodo, di tipo `T[]`.

Possiamo pertanto definire un metodo generico, con un tipo parametro `T`, in grado di ordinare qualunque array di oggetti di tipo `T`. Per garantire che il tipo argomento fornito in corrispondenza di `T` sia “ordinabile”, richiediamo che `T` implementi `Comparable<T>` o, più in generale, secondo quanto già discusso nei Paragrafi 6.17 e 9.3, che `T` sia un sottotipo di `Comparable<S>`, dove `S` è supertipo di `T`:

```
public static <T extends Comparable<? super T>> void ordina(T[] a) {
    T temp;
    boolean scambiato;
    do {
        scambiato = false;
        for (int i = 1; i < a.length; i++)
            if (a[i - 1].compareTo(a[i]) > 0) {
                temp = a[i - 1];
                a[i - 1] = a[i];
                a[i] = temp;
                scambiato = true;
            }
    } while (scambiato);
}
```

```

        a[i] = temp;
        scambiato = true;
    }
} while (scambiato);
}

```

Supponiamo di aver dichiarato una variabile di tipo array di **Frazione**:

```
Frazione[] frazioni;
```

di averle assegnato un array di frazioni e di averlo “riempito” di frazioni assegnando un riferimento a una frazione a ogni posizione dell’array. Dato che la classe **Frazione** implementa l’interfaccia **Comparable<Frazione>**, per ordinare l’array possiamo utilizzare la chiamata

```
GestioneArray.ordina(frazioni)
```

Per concludere questo paragrafo riportiamo il testo di un’applicazione che legge una sequenza di 10 frazioni, la ordina utilizzando il metodo appena definito e la visualizza:

```

import prog.io.*;
import prog.utili.Frazione;
import prog.utili.GestioneArray;

class OrdinaFrazioni {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        final int MAX = 10;
        Frazione[] frazioni = new Frazione[MAX];

        //fase lettura
        for (int pos = 0; pos < MAX; pos++) {
            out.print("Lettura della frazione " + (pos + 1));
            int num = in.readInt("Numeratore: ");
            int den = in.readInt("Denominatore: ");
            frazioni[pos] = new Frazione(num,den);
        }

        //ordinamento dell’array
        GestioneArray.ordina(frazioni);

        //fase di scrittura
    }
}

```

```

    for (int pos = 0; pos < frazioni.length; pos++)
        out.println(frazioni[pos].toString());
}
}

```

Ricerca dicotomica

Dopo avere illustrato una tecnica per ordinare un array, indichiamo ora una tecnica per individuare un elemento all'interno di un array. Se l'array non è ordinato, occorre esaminare i suoi elementi uno dopo l'altro, fino a individuare l'elemento cercato o raggiungere la fine dell'array. Se l'array è ordinato, è possibile utilizzarne una tecnica molto più veloce detta *ricerca binaria* o *dicotomica*. L'algoritmo che si utilizza in questo tipo di ricerca è lo stesso che si usa comunemente per trovare una parola in un dizionario: anziché scorrere tutti i termini alla ricerca di quello desiderato, si inizia la ricerca aprendo il dizionario più o meno a metà e si osservano le parole presenti nella pagina selezionata. Salvo che si sia molto fortunati e si trovi subito la parola cercata, si possono verificare due situazioni:

- la parola cercata precede, in ordine alfabetico, quelle della pagina selezionata: in questo caso si ripete la ricerca nella prima metà del dizionario utilizzando la stessa tecnica
- la parola cercata segue, in ordine alfabetico, quelle della pagina selezionata: in questo caso si ripete la ricerca nella seconda metà del dizionario utilizzando la stessa tecnica.

Ripetendo la ricerca si vanno a esaminare porzioni di dizionario sempre più ristrette, fino a individuare la pagina desiderata.

Proviamo a descrivere tale procedimento mediante un algoritmo:

```

    considera l'intero dizionario
    while (la parte che stai considerando ha almeno due pagine) {
        seleziona la pagina centrale della parte considerata
        if (in ordine alfabetico le parole della pagina centrale precedono quella cercata)
            considera la parte a destra della pagina centrale
        else if (in ordine alfabetico le parole della pagina centrale seguono quella cercata)
            considera la parte a sinistra della pagina centrale
        else
            considera solo la pagina centrale
    }

```

Supponiamo ora di disporre, anziché di un dizionario, di un array a di stringhe. In luogo di una pagina di dizionario consideriamo un elemento dell'array. Per delimitare la parte di array di volta in volta considerata, utilizziamo due indici: `sx` e `dx`. Inizialmente gli indici individuano la prima e l'ultima posizione dell'array:

```
int sx = 0, dx = a.length - 1;
```

La condizione del ciclo `while` è `sx < dx`. Per selezionare l'elemento di mezzo della porzione considerata, calcoliamo la media dei due indici `sx` e `dx`:

```
m = (sx + dx) / 2;
```

Possiamo confrontare alfabeticamente le stringhe tramite il metodo `compareTo`. Ad esempio, per verificare che la parola in posizione `m` preceda quella cercata è sufficiente eseguire il metodo `compareTo` di `a[m]` fornendo come argomento la parola cercata e controllando che il risultato sia minore di zero. In caso affermativo, per considerare la parte destra dell'array basta spostare l'indice `sx`. Ecco il codice del ciclo, in cui `chiave` è un riferimento di tipo `String` contenente la parola cercata:

```
while (sx < dx) {
    m = (sx + dx) / 2;
    if (a[m].compareTo(chiave) < 0)
        sx = m + 1;
    else if (a[m].compareTo(chiave) > 0)
        dx = m - 1;
    else
        sx = dx = m;
}
```

Alla fine del ciclo è sufficiente controllare l'elemento di `a` in posizione `sx`: se coincide con `chiave`, allora è stata individuata la posizione dell'elemento, altrimenti l'elemento non c'è.

```
if (a[sx].compareTo(chiave) == 0)
    return sx;
else return -1;
```

Osserviamo che, a ogni iterazione del precedente algoritmo, la dimensione della parte di array in cui cercare si riduce almeno a metà. Se inizialmente si considera un array di 1000 elementi, dopo la prima iterazione si prenderà in considerazione una sua parte, di 500 elementi al massimo; dopo la seconda iterazione una parte di non più di 250 elementi, e così via, dimezzando ogni volta. Alla decima iterazione ci troveremo a considerare al massimo due elementi e, quindi, al passo successivo, individueremo sicuramente l'elemento desiderato. Per 1000 elementi, dunque, il numero di ripetizioni del ciclo è tutt'al più 11. Utilizzando invece una ricerca sequenziale, occorrerebbe effettuare un ciclo ripetuto, nel caso peggiore, 1000 volte. Con 100000 elementi, il numero di ripetizioni del ciclo in una ricerca sequenziale è circa 100000; con una ricerca dicotomica è al massimo 18. In generale, se l'array è di n elementi, la ricerca sequenziale richiede n iterazioni, quella dicotomica circa $\log_2 n$ iterazioni. Quando l'array è ordinato, è sicuramente preferibile quest'ultimo tipo di ricerca, decisamente più veloce.

L'unico metodo della classe `String` che abbiamo utilizzato per scrivere il metodo `cerca` è il metodo `compareTo`, comune a tutte le classi che implementano l'interfaccia `Comparable`. In effetti, se dovessimo cercare un numero all'interno di un elenco ordinato, potremmo utilizzare esattamente la stessa tecnica.

Possiamo dunque scrivere un metodo generico per ricercare un elemento in un array di oggetti il cui tipo implementi l'interfaccia Comparable:

```
public static <T extends Comparable<? super T>>
    int cerca(T[] a, T chiave) {
    int sx = 0, dx = a.length - 1, m;

    while (sx < dx) {
        m = (sx + dx) / 2;
        if (a[m].compareTo(chiave) < 0)
            sx = m + 1;
        else if (a[m].compareTo(chiave) > 0)
            dx = m - 1;
        else
            sx = dx = m;
    }
    if (a[sx].compareTo(chiave) == 0)
        return sx;
    else return -1;
}
```

In realtà, l'oggetto da cercare potrebbe anche essere di un supertipo S di T, purché gli oggetti di T siano confrontabili con esso.

Quindi, il metodo può essere reso più generale modificando l'intestazione, nella quale si indicano due tipi parametrali come segue:

```
public static <S extends Comparable<S>, T extends S>
    int cerca(T[] a, S chiave)
```

Ecco un esempio d'uso del metodo appena definito nel caso di una sequenza di frazioni:

```
import prog.io.*;
import prog.utili.Frazione;
import prog.utili.GestioneArray;

class TrovaFrazione {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        final int MAX = 10;
        Frazione[] frazioni = new Frazione[MAX];
```

```

//lettura della sequenza di frazioni
for (int pos = 0; pos < MAX; pos++) {
    out.print("Lettura della frazione " + (pos + 1));
    int num = in.readInt("Numeratore: ");
    int den = in.readInt("Denominatore: ");
    frazioni[pos] = new Frazione(num,den);
}

//ordinamento dell'array
GestioneArray.ordina(frazioni);

//ricerca della frazione
Frazione daCercare;
out.print("Inserisci la frazione da cercare");
int num = in.readInt("Numeratore: ");
int den = in.readInt("Denominatore: ");
daCercare = new Frazione(num,den);
int risultato = GestioneArray.cerca(frazioni, daCercare);

//fase di scrittura
if (risultato == -1)
    out.println("La frazione " + daCercare.toString() +
                " non appartiene alla sequenza");
else
    out.println("La frazione " + daCercare.toString() +
                " è in posizione " + risultato);
}
}

```

Esercizi

- 9.14 Modificate l'applicazione TrovaFrazione in modo che, prima della lettura delle frazioni, richieda all'utente quante frazioni vuole inserire. Dopo la fase di lettura, l'applicazione dovrà visualizzare l'elenco delle frazioni inserite, in ordine di inserimento, e poi in ordine crescente. Infine, l'applicazione dovrà permettere all'utente di cercare una o più frazioni nell'elenco.
- 9.15 Progettate una classe che fornisca metodi statici utili per la gestione di matrici (array di array) di valori di tipo `int`. Tra le funzionalità che dovranno essere offerte vi sono: calcolo della somma di ciascuna riga, calcolo della somma di ciascuna colonna, calcolo del minimo di ciascuna riga, calcolo del minimo di ciascuna colonna, calcolo della somma dell'intera

matrice, calcolo del minimo dell'intera matrice. Ad esempio, il metodo che calcola la somma di ciascuna riga riceverà come argomento il riferimento a una matrice e restituirà un array contenente le somme degli elementi di ciascuna riga. Scrivete poi una classe di prova per i vari metodi.

- 9.16 Nella classe realizzata per l'esercizio precedente, aggiungete un metodo generico che riceva come argomento una matrice di elementi di un tipo T, dove T è un tipo parametro che implementa l'interfaccia Comparable: il metodo deve restituire l'elemento minimo all'interno della matrice.

Aggiungete poi un ulteriore metodo generico che calcoli il minimo elemento di ogni riga in una matrice di tipo T. Tale metodo dovrà ricevere due argomenti: il primo è il riferimento alla matrice, il secondo è il riferimento a un array di elementi di tipo T, di lunghezza pari al numero di righe della matrice. Il metodo dovrà porre il risultato nell'array fornito tramite il secondo argomento. (Non è possibile costruire array di un tipo parametro, cioè non è possibile scrivere new T[...]. Per questa ragione l'array in cui porre il risultato deve essere fornito dal chiamante).

Aggiungete un metodo analogo per il calcolo della somma delle colonne. Scrivete poi delle applicazioni di prova per questi metodi.

- 9.17 Scrivete un metodo che, ricevendo il riferimento a una matrice di qualunque tipo, controlli se è simmetrica (restituendo la risposta sotto forma di boolean), cioè controlli se è quadrata e, per ogni coppia di indici i e j, se l'elemento di posto i, j è uguale a quello di posto j, i (ricordatevi di utilizzare, per i confronti, il metodo equals).
- 9.18 Scrivete dei metodi per il calcolo di operazioni su matrici di interi quali la somma e il prodotto di due matrici e la trasposizione di una matrice.

Organizzazione della memoria e ricorsione

In questo capitolo studieremo alcuni importanti aspetti relativi all'organizzazione e alla gestione della memoria da parte della Java Virtual Machine durante l'esecuzione delle applicazioni. Mostreremo poi come la strategia utilizzata per gestire la memoria relativa ai metodi permetta di realizzare la ricorsione, cioè la possibilità di un metodo di richiamare se stesso.

Gli argomenti trattati in questo capitolo possono essere adattati facilmente a molti linguaggi di programmazione, object oriented e no. In particolare, per la trattazione di questi argomenti, le caratteristiche object oriented di Java sono irrilevanti. Al fine di focalizzarci sugli argomenti specifici indicati nel titolo del capitolo, abbiamo scelto apposta esempi sostanzialmente svincolati dalla natura a oggetti del linguaggio Java.

10.1 Invocazione di metodi, passaggio di parametri e rientro

Come abbiamo visto nei capitoli precedenti, un metodo è una porzione di codice cui si associa un nome. L'intestazione del metodo fornisce una serie di informazioni sul modo in cui può essere utilizzato. In particolare ci sono le seguenti.

- Alcuni *modificatori*, come ad esempio `static` e `public`, che forniscono alcune informazioni relative al metodo e alla sua fruibilità da parte di altri metodi o classi. I modificatori possono anche non esserci.
- Un *nome di tipo*, che indica il tipo restituito dal metodo; questo può essere un tipo primitivo o un tipo riferimento. In quest'ultimo caso, il metodo restituisce un riferimento a un oggetto di tale tipo (oppure il riferimento `null`). Ad esempio, se il tipo indicato è il nome di una classe, il metodo restituisce un riferimento a un'istanza di tale classe (ricordiamo che le istanze di tutte le sottoclassi di una classe C sono istanze di C). Per indicare che un metodo non restituisce alcun valore si utilizza la parola `void`.
- Il *nome del metodo*, che è un identificatore scelto da chi ha scritto il metodo.

- La *lista dei parametri* del metodo, che è racchiusa tra parentesi tonde. I parametri della lista (detti anche *parametri formali*) sono identificatori scelti da chi ha scritto il metodo. Ogni identificatore è preceduto dal nome del proprio tipo. I parametri sono separati tra loro da virgolette.

Dopo la lista dei parametri possono esserci altre informazioni, che esamineremo nel prossimo capitolo, relative alle *eccezioni* che il metodo può sollevare.

Consideriamo ad esempio il seguente metodo contenuto nella classe `Frazione`. Dati due interi positivi, esso restituisce come risultato il loro massimo comun divisore:

```
private static int mcd(int a, int b) {
    int resto;
    do {
        resto = a % b;
        a = b;
        b = resto;
    } while (resto != 0);
    return a;
}
```

I modificatori sono `private` e `static`, il tipo del risultato è `int`, il nome del metodo è `mcd` e ci sono due parametri di tipo `int`.

Un metodo viene invocato utilizzandone il nome (eventualmente preceduto da un riferimento all'oggetto cui il metodo appartiene, come ad esempio i metodi della classe `String`, o dal nome della classe cui appartiene, come ad esempio i metodi statici della classe `Math` o della classe `Character`) seguito dalla lista degli *argomenti* (o *parametri attuali*) con cui il metodo viene chiamato. Gli argomenti devono corrispondere in nome e in tipo ai parametri della segnatura del metodo. Ad esempio al metodo `mcd` scritto sopra dovranno essere forniti come argomenti due valori di tipo `int` (o meglio, due valori che possano essere assegnati al tipo `int`).

Più precisamente, quando si esegue l'*invocazione* di un metodo, la prima operazione che viene effettuata è il *passaggio dei parametri*. Ogni parametro scritto nell'intestazione del metodo è di fatto una variabile locale del metodo. Al momento della chiamata, tale variabile viene inizializzata con il *valore* dell'argomento corrispondente. Questa modalità di passaggio, l'unica disponibile nel linguaggio Java, è detta *passaggio per valore*.

Ad esempio, nella chiamata

```
mcd(x + y, z)
```

al parametro `a` del metodo `mcd` viene assegnato il risultato dell'espressione `x + y`, mentre al parametro `b` viene assegnato il valore contenuto nella variabile `z`. Chiaramente, come per gli assegnamenti, tali valori devono essere di tipo `int` o di un tipo che possa essere promosso automaticamente a `int`. Dunque, se la variabile `z` è di tipo `double`, la chiamata non è corretta e il compilatore segnalerà un errore.

Dopo il passaggio dei parametri inizia l'esecuzione del metodo, che avviene nella solita maniera, utilizzando cioè le variabili definite all'interno del metodo, i parametri del metodo ed

eventuali variabili che il metodo eredita dall'ambiente nel quale è definito (i campi della stessa classe del metodo).

L'esecuzione del metodo termina con l'istruzione `return`, che provoca la *restituzione del risultato* e il *rientro* al codice chiamante. Il risultato restituito dal metodo viene indicato mediante un'espressione subito dopo la parola `return`. Il tipo di questa espressione dev'essere un tipo assegnabile al tipo restituito dal metodo (ad esempio, se il tipo restituito dal metodo è `long`, il tipo dell'espressione può essere `int`, ma non `double`). Le variabili definite all'interno del metodo (compresi i parametri) vengono distrutte, e l'esecuzione riprende dal punto in cui è avvenuta la chiamata, con l'utilizzo del valore restituito dal metodo.

Si supponga, ad esempio, che prima di eseguire l'istruzione

```
x = 3 * mcd(x + y, z) + 1;
```

le variabili `x`, `y` e `z` contengano, rispettivamente, i valori 6, 4 e 4. Per eseguire l'istruzione occorre calcolare prima di tutto il lato destro dell'assegnamento che, a sua volta, richiede l'invocazione del metodo `mcd`. Al momento della chiamata del metodo, ai parametri `a` e `b` vengono assegnati i valori 10 e 4 dei rispettivi argomenti. Inizia dunque l'esecuzione del metodo e, dopo un certo numero di iterazioni, viene raggiunta l'istruzione `return a`, con `a` contenente il valore 2. A questo punto c'è il rientro. Tutti i dati "locali" del metodo (cioè i parametri `a` e `b`, e la variabile `resto`) vengono eliminati dalla memoria. Il risultato restituito, cioè 2, può essere utilizzato per completare l'esecuzione dell'istruzione calcolando il valore del lato destro dell'assegnamento, cioè 7, e assegnandolo alla variabile `x`.

L'esecuzione di un metodo *dove sempre terminare con l'istruzione `return`*, con l'unica eccezione dei metodi che hanno come tipo restituito `void`, la cui esecuzione può terminare, senza istruzione `return`, alla fine del codice del metodo.

In molti linguaggi di programmazione (a oggetti o no) è possibile definire porzioni di codice strutturate in maniera simile ai metodi: i *sottoprogrammi*. I sottoprogrammi che restituiscono un valore sono detti *funzioni*, quelli che non restituiscono nulla vengono chiamati *procedure*. Oltre al passaggio di parametri per valore, che, lo ripetiamo, è l'unico messo a disposizione dal linguaggio Java, esistono altre modalità, come il passaggio per *riferimento* (detto anche per *variabile* o per *indirizzo*), per *nome*, per *valore/risultato*.

La caratteristica principale del passaggio per valore è la totale indipendenza fra parametro attuale (quello fornito come argomento al momento dell'invocazione del metodo) e parametro formale (quello che compare nell'intestazione del metodo) dal momento in cui il valore del parametro attuale viene copiato nel parametro formale. Come abbiamo detto in precedenza nel passaggio per valore, il parametro formale è a tutti gli effetti una variabile locale. Al momento della chiamata, il parametro attuale viene valutato e il suo valore viene copiato in questa variabile locale. Da tale momento in poi non vi è più alcuna relazione fra parametro attuale e parametro formale. Consideriamo il seguente metodo:

```
public static void f (int x) {
    x = 1;
}
```

e supponiamo di eseguire le seguenti istruzioni

```
int y = 0;
f(y);
```

Quello che accade all'interno del metodo non influenza in alcun modo il valore della variabile *y*; quindi il suo valore risulta invariato dopo l'esecuzione del metodo, a prescindere dagli assegnamenti eseguiti sulla variabile *x* all'interno del metodo.

Mostriamo ora un esempio di metodo che riceve un riferimento come parametro. Consideriamo la seguente classe A contenente solo un campo:

```
public class A {
    int x;
}
```

e il seguente metodo:

```
public static void g1(A alfa) {
    alfa.x = 1;
}
```

Supponiamo che *beta* sia un riferimento a un oggetto di tipo A in cui il valore del campo *x* è 0 e consideriamo l'invocazione del metodo *g1* con argomento *beta*, cioè *g1(beta)*. In questo caso, il valore dell'argomento utilizzato nella chiamata è il riferimento *beta*. Come abbiamo detto, tale valore viene copiato nel parametro formale *alfa*. Durante l'esecuzione del metodo, *alfa* si riferirà dunque al *medesimo* oggetto cui si riferisce *beta*. Pertanto l'assegnamento *alfa.x = 1* all'interno del metodo andrà a modificare tale oggetto. In altre parole, dopo l'esecuzione del metodo sarà possibile osservare un effetto sull'oggetto riferito da *beta* (nonostante il riferimento *beta* non abbia subito cambiamenti). I metodi possono quindi modificare gli oggetti grazie al passaggio dei riferimenti.¹

Si consideri ora il metodo

```
public static void g2(A alfa) {
    alfa = new A();
    alfa.x = 1;
}
```

e la chiamata *g2(beta)*. Il lettore può facilmente verificare che, in questa circostanza, l'oggetto cui fa riferimento *beta* non viene modificato dal metodo.

Consideriamo ora il caso degli array. Ricordiamo che in Java gli array sono oggetti, mentre le variabili di tipo array sono riferimenti a essi. Di conseguenza, anche in questo caso è possibile scrivere metodi che ricevano come parametro variabili di tipo array e modifichino l'oggetto

¹ Nel passaggio per riferimento o per variabile, presente in altri linguaggi di programmazione, i sottoprogrammi possono modificare direttamente le variabili fornite tramite gli argomenti. Nel linguaggio Java questo è impossibile. Tuttavia, come mostrato nell'esempio, passando come argomenti (per valore) dei riferimenti, è possibile comunque produrre modifiche sugli oggetti.

array (una situazione tipica può essere un metodo di ordinamento che riceve come argomento il riferimento all'array da ordinare).

Presentiamo due esempi analoghi ai precedenti. Consideriamo il metodo:

```
public static void h1(int[] alfa) {
    alfa[2] = 1;
}
```

e le istruzioni

```
int[] beta = {0,0,0};
h1(beta);
```

Dato che gli array sono oggetti, al momento dell'esecuzione di `h1(beta)`, il valore di `beta`, che è un riferimento all'array appena costruito, viene copiato in `alfa`. Quindi l'assegnamento all'interno del metodo è un assegnamento alla posizione 2 dell'array cui fa riferimento `beta` che, dopo l'esecuzione del metodo, risulta quindi modificato.

Al contrario, nel caso del metodo:

```
public static void h2(int[] alfa) {
    alfa = new int[3];
    alfa[2] = 1;
}
```

e delle istruzioni:

```
int[] beta = {0,0,0};
h2(beta);
```

l'array cui fa riferimento `beta` non risulta modificato dopo l'esecuzione del metodo `h2`, in quanto il riferimento contenuto in `alfa` dopo l'esecuzione di `alfa = new int[3]` è diverso da quello contenuto in `beta`.

10.2 Organizzazione della memoria

Esaminiamo ora come è organizzata la memoria utilizzata dalla Java Virtual Machine durante l'esecuzione. Ci sono tre aree principali.

- *Memoria statica.*

È utilizzata per i campi statici delle classi. Quando una o più classi vengono caricate per l'esecuzione, si riserva una porzione di memoria ai campi statici. Tali campi sono noti *prima* dell'esecuzione e possono essere impiegati durante tutta l'esecuzione. La quantità di memoria statica necessaria per una classe può essere stabilita a priori esaminando esclusivamente il testo della classe.

- *Stack.*

È utilizzato per contenere i dati usati dai singoli metodi che vengono man mano eseguiti. La sua struttura evolve dinamicamente durante l'esecuzione in base alle chiamate dei metodi. Esamineremo in dettaglio quest'area tra poco.

- *Heap.*

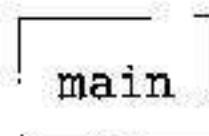
È utilizzato per memorizzare gli oggetti creati dinamicamente richiamando i costruttori all'interno di espressioni `new`. Quando un oggetto non è più accessibile, cioè quando non ci sono più riferimenti a esso, l'area di memoria utilizzata dall'oggetto può essere recuperata e riutilizzata successivamente per altri oggetti. Questa funzionalità viene svolta automaticamente dalle routine di *garbage collection* (in altri linguaggi, la garbage collection dev'essere gestita direttamente dal programmatore). Con la creazione e distruzione di molti oggetti si ha il fenomeno della *frammentazione*: nello heap possono esserci molte aree di piccola dimensione che, pur essendo libere, sono difficilmente utilizzabili in quanto separate tra loro da aree occupate da oggetti. Le routine di garbage collection si occupano anche della riorganizzazione dello heap: quando c'è molta frammentazione, tutti gli oggetti vengono spostati in zone adiacenti, in modo che le aree libere possano essere ricompattate in un'unica, grande area libera.

Analizziamo più in dettaglio l'area di memoria utilizzata per i dati dei metodi, cioè lo stack. Quest'area prende il nome dalla struttura dati utilizzata per gestirla, lo *stack* appunto (pila). In questo tipo di struttura è possibile aggiungere o eliminare elementi solo in cima alla pila (si pensi a una pila di piatti o a una pila di libri). La memoria stack non è altro che una *pila* di *record di attivazione*, dove ogni record di attivazione è un'area di memoria locale contenente i dati relativi a ciascun metodo attivato.

All'inizio dell'esecuzione del programma, lo stack contiene il record di attivazione del metodo `main` da cui prende avvio l'esecuzione. Quando viene chiamato un metodo, in cima allo stack è aggiunto un record di attivazione per il metodo chiamato. Tale record viene distrutto al rientro dal metodo (si noti che il primo metodo che può terminare è sempre l'ultimo a essere stato chiamato).²

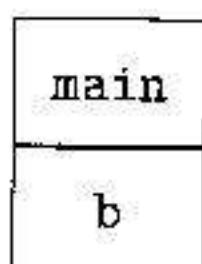
Consideriamo ad esempio un metodo `main` che richiama un metodo `b`, che a sua volta richiama un metodo `a`. Disegniamo l'organizzazione dello stack in record di attivazione durante l'esecuzione. Per comodità rappresenteremo sempre lo stack al contrario, ponendo cioè l'elemento corrispondente alla cima dello stack in basso.

All'inizio dell'esecuzione lo stack conterrà solamente il record di attivazione di `main`:

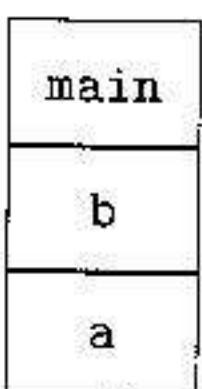


² In quest'analisi menzioniamo esplicitamente i metodi, sottintendendo però anche i costruttori, per i quali i meccanismi di attivazione, il passaggio dei parametri e la disattivazione sono del tutto analoghi. In particolare, anche alla chiamata di un costruttore verrà creato un corrispondente record di attivazione sullo stack, che sarà distrutto al termine dell'esecuzione del costruttore.

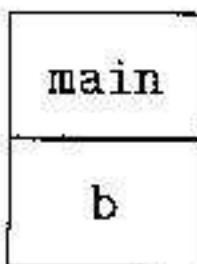
Quando `main` richiama `b`, in cima allo stack verrà posto un nuovo record di attivazione:



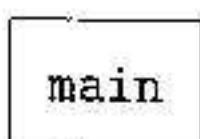
Se `b` a un certo punto richiama `a`, la struttura dello stack diventerà:



Quando si rientra da `a` eseguendo un'istruzione `return`, il record di attivazione di `a` viene distrutto e si riprende l'esecuzione dal punto, in questo caso in `b`, dove è avvenuta la chiamata. Pertanto la struttura dello stack torna a essere:



Infine, al termine dell'esecuzione di `b`, anche il record di attivazione di `b` viene eliminato dallo stack:



Nel record di attivazione che si trova in cima allo stack (che chiameremo *record di attivazione corrente*) si trovano in ogni istante i dati locali del metodo in esecuzione.

Struttura del record di attivazione

Abbiamo visto che il *record di attivazione* di un metodo `m` è la struttura di memoria associata a ciascuna attivazione di `m`. Tale struttura contiene:

(1) Informazioni relative all'attivazione del metodo.

Queste informazioni vengono impiegate per:

- ricevere i risultati dei metodi richiamati da `m`
- effettuare correttamente il *rientro* dai metodi richiamati da `m`.

(2) Dati del metodo.

Nel record di attivazione trovano posto i dati dichiarati localmente al metodo, cioè:

- le *variabili locali*
- i *parametri per valore*, inizializzati con gli argomenti specificati al momento della chiamata.

Evoluzione dello stack durante l'esecuzione

Mostriremo ora l'evoluzione dello stack durante l'esecuzione. Nella presentazione utilizziamo come esempio la seguente applicazione:

```
import prog.io.*;

public class Doppio {

    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int x = in.readInt("Inserire un intero ");
        int y = doppio(x); //1
        int z = doppio(y); //2
        out.println(y);
        out.println(z);
    }

    public static int doppio(int n) {
        int k = 2 * n;
        return k;
    }
}
```

Si osservi che la classe contiene un metodo statico `doppio`, che restituisce il doppio del valore ricevuto tramite il parametro. Tale metodo viene richiamato due volte dal metodo `main`, calcolando così il doppio e il quadruplo di ciò che è stato fornito in ingresso. Per comodità le due chiamate di `doppio` sono state marcate con i commenti //1 e //2.

All'inizio dell'esecuzione lo stack contiene solo il record di attivazione del metodo `main`, in cui è incluso lo spazio per le variabili locali del metodo (per brevità non rappresentiamo i riferimenti `in` e `out` utilizzati per la lettura e per la scrittura, e il parametro `args`, il cui valore non è utilizzato nell'esempio) e per le informazioni utili all'esecuzione, e cioè i campi "risultato" e "ritorno" impiegati, rispettivamente, per ricevere i risultati dai metodi chiamati e per memorizzare il punto da cui riprendere l'esecuzione al rientro da un metodo chiamato. Il punto interrogativo

presente nei vari campi del record di attivazione indica che non è ancora stato assegnato loro alcun valore:

main	
x	?
y	?
z	?
risultato	?
ritorno	?

Supponiamo di inserire il valore 3. Dopo l'assegnamento a x, il contenuto dello stack sarà:

main	
x	3
y	?
z	?
risultato	?
ritorno	?

L'istruzione successiva è un assegnamento sul cui lato destro c'è una chiamata al metodo doppio. Alla *chiamata* di un metodo vengono effettuate le seguenti operazioni:

- memorizzazione, nel record di attivazione *del chiamante*, del punto di rientro (campo ritorno)
- creazione del record di attivazione del metodo chiamato
- passaggio dei parametri, cioè valutazione e assegnamento degli argomenti ai parametri formali.

Ecco il contenuto dello stack dopo queste operazioni:

main	
x	3
y	?
z	?
risultato	?
ritorno	//1

doppio	
n	3
k	?
risultato	?
ritorno	?

A questo punto inizia l'esecuzione del codice del metodo. Dopo l'assegnamento alla variabile k, il contenuto dello stack diventa:

main	
x	3
y	?
z	?
risultato	?
ritorno	//1

doppio	
n	3
k	6
risultato	?
ritorno	?

L'istruzione successiva è `return k`, con cui il metodo restituisce al chiamante il risultato e termina la propria esecuzione. In questa fase, detta *rientro dal metodo*, vengono effettuate in particolare le seguenti operazioni:

- memorizzazione del risultato nel record di attivazione del chiamante
- distruzione del record di attivazione in cima alla pila
- proseguimento dell'esecuzione dal punto indicato nel record di attivazione del chiamante.

Dunque, dopo avere eseguito queste operazioni, il contenuto dello stack sarà:

main	
x	3
y	?
z	?
risultato	6
ritorno	//1

A questo punto il metodo main può completare l'operazione di assegnamento alla variabile y:

main	
x	3
y	6
z	?
risultato	6
ritorno	//1

L'istruzione successiva è una nuova invocazione del metodo doppio. Pertanto sullo stack verrà creato un *nuovo* record di attivazione per il metodo doppio. In particolare, dopo il passaggio dei parametri, il contenuto dello stack sarà:

main	
x	3
y	6
z	?
risultato	6
ritorno	//2

doppio	
n	6
k	?
risultato	?
ritorno	?

Sottolineiamo il fatto che il record di attivazione creato per il metodo `doppio` è nuovo e non ha niente a che vedere con quello creato per l'attivazione precedente. Soprattutto, come evidenziato nella figura, il contenuto della variabile `k` non è definito, mentre quello del parametro `n` corrisponde al valore fornito come argomento nell'ultima chiamata.

Dopo l'esecuzione dell'assegnamento a `k`, il contenuto dello stack diventa:

main	
x	3
y	6
z	?
risultato	6
ritorno	//2

doppio	
n	6
k	12
risultato	?
ritorno	?

L'istruzione successiva è quella di rientro, che assegnerà il risultato 12 al campo `risultato` del chiamante. Dopo la distruzione del record di attivazione, l'esecuzione riprende dal punto memorizzato nel campo `ritorno`, in questo caso l'istruzione marcata con il commento `//2`. Dopo avere eseguito l'assegnamento, il contenuto dello stack sarà:

main	
x	3
y	6
z	12
risultato	12
ritorno	//2

Le istruzioni successive visualizzano i valori delle variabili `y` e `z`. Raggiunta l'ultima istruzione, anche il record di attivazione del metodo `main` viene distrutto. La Java Virtual Machine conclude così l'esecuzione dell'applicazione.

10.3 Metodi ricorsivi

In Java un metodo può richiamare se stesso. Questa possibilità è chiamata *ricorsione*; i metodi che la utilizzano sono detti *ricorsivi*. Presenteremo ora un esempio di metodo ricorsivo, evidenziandone l'esecuzione secondo il modello descritto nel paragrafo precedente.

In generale un'entità viene detta *ricorsiva* se è definita in termini di se stessa. In matematica si incontrano tipici esempi di entità definite ricorsivamente tramite le *definizioni per ricorrenza* o *definizioni induttive*. Ad esempio il *fattoriale* di un numero intero non negativo n , indicato con $n!$, può essere definito come:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n - 1)! & \text{altrimenti} \end{cases}$$

La definizione precedente contiene un *caso base* (corrispondente a $n = 0$) e un *caso induttivo* (corrispondente alla parola “altrimenti”) in cui la definizione di $n!$ viene data in termini di $(n - 1)!$. Utilizzando il caso induttivo otteniamo ad esempio la catena di uguaglianze:

$$3! = 3 \cdot 2! = 3 \cdot (2 \cdot 1!) = 3 \cdot (2 \cdot (1 \cdot 0!)).$$

A questo punto possiamo applicare il caso base sostituendo $0!$ con 1 e calcolare il risultato:

$$3 \cdot (2 \cdot (1 \cdot 1)) = 3 \cdot (2 \cdot 1) = 3 \cdot 2 = 6.$$

Data la definizione induttiva, è immediato scrivere un metodo Java **fattoriale** per il calcolo del fattoriale di n che ricalchi la definizione stessa:

- se il valore di n è zero, il metodo restituisce immediatamente 1 come risultato;
- negli altri casi il metodo chiama se stesso per il calcolo del fattoriale di $n - 1$ e, una volta ottenuto il risultato, lo moltiplica per n restituendo il valore ottenuto all'esterno, cioè a chi l'ha richiamato (si suppone che il metodo venga chiamato con un argomento non negativo).

Il metodo è dunque:

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

L'esecuzione del metodo avviene, come per tutti gli altri metodi, creando al momento della chiamata del metodo un record di attivazione per esso, che sarà distrutto al rientro. Quando il metodo richiamerà se stesso, verrà creato un nuovo record di attivazione dello stesso metodo. Nello stesso istante lo stack può dunque contenere più record di attivazione di un metodo ricorsivo.

Per evidenziare questi aspetti, inseriamo il metodo **fattoriale** all'interno di un'applicazione e simuliamone dinamicamente l'esecuzione nel caso venga inserito come input 3.

```

import prog.io.*;

public class Fattoriale {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int x = in.readInt("Inserire un intero non negativo ");
        while (x < 0)
            x = in.readInt("L'input non può essere negativo. " +
                           "Ripeti l'inserimento ");
        int f = fattoriale(x); //+
        out.println("Il fattoriale di " + x + " è " + f);
    }

    public static int fattoriale(int n) {
        if (n == 0)
            return 1;
        else
            return n * fattoriale(n - 1); /* */
    }
}

```

Ecco che cosa conterrà lo stack dopo la lettura del dato nel metodo `main`:

main	
x	3
f	?
risultato	?
ritorno	?

L'istruzione successiva è un assegnamento sul cui lato destro si trova un'invocazione del metodo `fattoriale` con argomento `x`. Pertanto viene creato sullo stack un record di attivazione per `fattoriale` dove, nello spazio per il parametro `n`, viene copiato il valore di `x`. Inoltre nel record di attivazione di `main` viene memorizzato il punto di ritorno, indicato nel testo da `//+`.

main	
x	3
f	?
risultato	?
ritorno	//+

fattoriale	
n	3
risultato	?
ritorno	?

L'esecuzione di **fattoriale** ha inizio valutando la condizione dell'**if**. Poiché risulta falsa, si esegue il ramo **else**, che contiene una chiamata al metodo **fattoriale** stesso.

Questa chiamata viene trattata nel solito modo: si memorizza il punto di ritorno nell'ambiente chiamante, si crea un nuovo record di attivazione in cima allo stack e si effettua il passaggio dei parametri. In particolare l'argomento nella chiamata è l'espressione **n - 1**. Nc viene calcolato il valore, cioè 2, nell'ambiente chiamante. Tale valore viene copiato nel parametro del nuovo ambiente, cioè nella variabile **n** del nuovo record di attivazione. Il contenuto dello stack diventa quindi:

main	
x	3
f	?
risultato	?
ritorno	//+

fattoriale	
n	3
risultato	?
ritorno	//*

fattoriale	
n	2
risultato	?
ritorno	?

Analogamente al caso precedente, l'esecuzione inizia valutando la condizione. Poiché è falsa, si procede a effettuare un'ulteriore chiamata ricorsiva ottenendo sullo stack:

main
x 3
f ?
risultato ?
ritorno //+
fattoriale
n 3
risultato ?
ritorno /*
fattoriale
n 2
risultato ?
ritorno /*
fattoriale
n 1
risultato ?
ritorno ?

L'esecuzione del codice del metodo richiede un'ulteriore chiamata ricorsiva. Dopo la chiamata, il contenuto dello stack è:

main
x 3
f ?
risultato ?
ritorno //+
fattoriale
n 3
risultato ?
ritorno /*
fattoriale
n 2
risultato ?
ritorno /*
fattoriale
n 1
risultato ?
ritorno /*
fattoriale
n 0
risultato ?
ritorno ?

Si comincia a eseguire il codice del metodo ancora una volta. In questo caso la condizione `n == 0` è vera. Pertanto viene eseguita l'istruzione `return 1`, con cui si scrive il valore 1 nell'ambiente chiamante, e si rientra dal metodo distruggendo il record di attivazione che si trova in cima allo stack. A questo punto il contenuto dello stack è:

main
x 3
f ?
risultato ?
ritorno //+
fattoriale
n 3
risultato ?
ritorno //*
fattoriale
n 2
risultato ?
ritorno //*
fattoriale
n 1
risultato 1
ritorno //*

Al rientro dal metodo si riprende l'esecuzione dal punto memorizzato nel campo `ritorno` del record di attivazione che si trova ora in cima allo stack. Usando il risultato ottenuto dalla chiamata ricorsiva, possiamo ora calcolare il valore dell'espressione `n * fatt(n - 1)` moltiplicando appunto il contenuto di `n` nel record di attivazione corrente per il risultato fornito dal metodo chiamato, memorizzato nel campo `risultato`. Il risultato dell'espressione è 1, che a sua volta viene restituito mediante l'istruzione `return n * fattoriale(n - 1)` all'ambiente esterno.

Dopo il rientro dal metodo, il contenuto dello stack è dunque:

main
x 3
f ?
risultato ?
ritorno //+
fattoriale
n 3
risultato ?
ritorno //*
fattoriale
n 2
risultato 1
ritorno //*

Si ricomincia di nuovo dal punto `//*` e si procede eseguendo l'istruzione

```
return n * fattoriale(n - 1)
```

mediante la quale si restituisce 2 all'ambiente esterno. Dopo il rientro dal metodo, il contenuto dello stack è:

main	
x	3
f	?
risultato	?
ritorno	//+
fattoriale	
n	3
risultato	2
ritorno	//*

Anche in questo caso il punto di rientro memorizzato è `//*`. Eseguendo l'istruzione

```
return n * fattoriale(n - 1)
```

si restituisce 6 all'ambiente esterno. Dopo il rientro dal metodo, il contenuto dello stack è:

main	
x	3
f	?
risultato	6
ritorno	//+

Si rientra ora al punto `//+`, dove si completa l'esecuzione dell'assegnamento

```
f = fattoriale(x)
```

copiando il valore restituito dal metodo (che si trova nel campo `risultato`) nella variabile `f`. Il contenuto dello stack diventa dunque:

main	
x	3
f	6
risultato	6
ritorno	//+

Esempio: stringhe palindrome

Riprendiamo l'esempio presentato nel Paragrafo 3.6, dove avevamo costruito un'applicazione per controllare se una stringa fosse palindroma.

Scriviamo ora un metodo con lo stesso compito. Il metodo ha la seguente intestazione:

```
public static boolean palindroma(String s)
```

Ricevendo come parametro la stringa da esaminare, il metodo deve restituire `true` se essa è palindroma, `false` in caso contrario. Utilizzando la tecnica del Paragrafo 3.6 potremmo far uso di un algoritmo *iterativo*, basando il metodo su un ciclo `for`. Svilupperemo invece una soluzione diversa: faremo uso della ricorsione.

A tale scopo cerchiamo di spiegare in maniera induttiva che cos'è una parola palindroma. Possiamo osservare che se il primo e l'ultimo simbolo di una parola sono differenti, allora la parola non è palindroma; se il primo e l'ultimo simbolo sono uguali, la parola è palindroma se e solo se la parola che si ottiene cancellando il primo e l'ultimo simbolo è ancora palindroma. Inoltre la parola vuota e tutte le parole costituite da un unico simbolo sono palindrome. Quindi:

- se la parola è di zero o un carattere, il risultato è `true`;
- altrimenti, se il primo e l'ultimo simbolo sono diversi, il risultato è `false`;
- altrimenti il risultato è lo stesso che si ha sulla parola ottenuta cancellando il primo e l'ultimo simbolo dalla parola data.

I primi due casi rappresentano la base della ricorsione, cioè i casi semplici, in cui possiamo fornire immediatamente la risposta; nel terzo caso, invece, la risposta si ottiene risolvendo lo stesso problema su un'istanza più semplice.

Ecco il metodo sviluppato secondo lo schema precedente:

```
public static boolean palindroma(String s) {
    int lung = s.length();
    if (lung <= 1)
        return true;
    else if (s.charAt(0) != s.charAt(lung - 1))
        return false;
    else return palindroma(s.substring(1, lung - 1));
}
```

Esercizi

10.1 La successione dei numeri di *Fibonacci* è data da 1, 1, 2, 3, 5, 8, 13, 21, ... In pratica il primo e il secondo numero della successione sono uguali a 1, mentre ognuno degli altri numeri è uguale alla somma dei due numeri che lo precedono.

Costruire sia in versione iterativa sia in versione ricorsiva un metodo che, ricevuto come parametro un numero k , restituisca il k -esimo termine della successione di Fibonacci. Scrivete poi un metodo `main` che richiami tale metodo e lo si simuli "manualmente" evidenziando il contenuto dello stack.

- 10.2 Scrivete tre metodi ricorsivi che, ricevuti due parametri interi non negativi x e y , restituiscano il valore di x elevato alla y . Il primo metodo dev'essere basato sulla seguente uguaglianza:

$$x^y = \begin{cases} x \cdot x^{y-1} & \text{se } y > 0, \\ 1 & \text{se } y = 0. \end{cases}$$

Il secondo e il terzo metodo trattano il caso di potenze pari e potenze dispari in modo diverso:

$$x^y = \begin{cases} x \cdot x^{y-1} & \text{se } y > 0 \text{ e } y \text{ è dispari,} \\ (x^{\frac{y}{2}})^2 & \text{se } y > 0 \text{ e } y \text{ è pari,} \\ 1 & \text{se } y = 0. \end{cases}$$

Per il calcolo di $(x^{\frac{y}{2}})^2$ nel secondo metodo, utilizzate due chiamate ricorsive che calcolano entrambe $x^{\frac{y}{2}}$, e calcolate quindi il prodotto dei risultati. Nel terzo metodo, invece, utilizzate una sola chiamata ricorsiva, che calcoli $x^{\frac{y}{2}}$ ponendo il risultato in una variabile che viene poi moltiplicata per se stessa per ottenere il quadrato.

Esaminate i tre metodi al fine di identificare, al variare di y , il numero totale di chiamate ricorsive (si consiglia di considerare prima il caso in cui y è potenza di 2). Verificate sperimentalmente il risultato ottenuto scrivendo una classe per la prova dei metodi in cui, a ogni chiamata ricorsiva, venga incrementato un contatore da stampare alla fine dell'esecuzione, allo scopo di conoscere il numero totale di chiamate effettuate (un modo semplice per realizzare il contatore consiste nell'utilizzare un campo statico).

- 10.3 Scrivete un metodo per il calcolo del massimo comun divisore tra due numeri non negativi basato sulla seguente uguaglianza:

$$\text{mcd}(x, y) = \begin{cases} x & \text{se } y = 0 \\ \text{mcd}(y, x \bmod y) & \text{altrimenti.} \end{cases}$$

- 10.4 Scrivete un metodo che riceva un parametro intero n e restituiscia il valore $f(n)$, dove f è la funzione definita come:

$$f(n) = \begin{cases} 1 & \text{se } n = 0 \\ n * (f(n - 1) + 1) & \text{se } n > 0. \end{cases}$$

- 10.5 Scrivete un metodo che riceva un parametro intero n e restituiscia il valore $f(n)$, dove f è la funzione definita come:

$$f(n) = \begin{cases} 0 & \text{se } n = 0 \\ n + f(n - 1) & \text{se } n > 0. \end{cases}$$

- 10.6 Scrivete un metodo che riceva due parametri interi x e y , e restituiscia il valore $f(x, y)$, dove f è la funzione definita come:

$$f(x, y) = \begin{cases} 1 & \text{se } y = 0 \\ 3 + f(y - 1, x) & \text{se } y > 0. \end{cases}$$

- 10.7 Scrivete un metodo che riceva due parametri interi x e y , e restituisca il valore $f(x, y)$, dove f è la funzione definita come:

$$f(x, y) = \begin{cases} 0 & \text{se } y = 0 \\ x + f(x, y - 1) & \text{se } y > 0. \end{cases}$$

- 10.8 Indicate l'output prodotto dall'esecuzione del metodo `main` della seguente classe qualora si fornisca come dato il numero 4:

```
import prog.io.*;

public class C {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();
        int x = in.readInt();
        int y = f(x);
        out.println(y);
    }

    public static int f(int n) {
        if (n == 0)
            return 1;
        else
            return n * (f(n - 1) + 1);
    }
}
```

- 10.9 Indicate l'output prodotto dall'esecuzione del metodo `main` della seguente classe qualora si fornisca come dato il numero 8:

```
import prog.io.*;

public class C {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();
        int x = in.readInt();
        int y = f(x);
```

```

        out.println(y);
    }

    public static int f(int n) {
        if (n == 1)
            return 1;
        else
            return n + f(n / 2);
    }
}

```

- 10.10 Indicate l'output prodotto dall'esecuzione del metodo `main` della seguente classe qualora si fornисca come dato il numero 7:

```

import prog.io.*;

public class C {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();
        int x = in.readInt();
        int y = f(x);
        out.println(y);
    }

    public static int f(int n) {
        if (n <= 1)
            return 1;
        else
            return 3 * f(n / 2) + 4 * f(n % 2);
    }
}

```

- 10.11 Indicate l'output prodotto dall'esecuzione del metodo `main` della seguente classe qualora si forniscano come dati i numeri 3 e 2:

```

import prog.io.*;

public class C {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();
        int x = in.readInt();

```

```
    int y = in.readInt();
    y = f(x, y);
    out.println(y);
}

public static int f(int m, int n) {
    if (n == 0)
        return 1;
    else
        return 3 + f(n - 1, m);
}
```

- 10.12 Indicate l'output prodotto dall'esecuzione del metodo `main` della seguente classe qualora si forniscano come dati i numeri 3 e 2:

```
import prog.io.*;

public class C {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();
        int x = in.readInt();
        int y = in.readInt();
        y = f(x, y);
        out.println(y);
    }

    public static int f(int m, int n) {
        if (n == 0)
            return 1;
        else
            return 3 + f(m - 1, n);
    }
}
```

- 10.13 Indicate l'output prodotto dall'esecuzione del metodo `main` della seguente classe qualora si forniscano come dati i numeri 5 e 3:

```
import prog.io.*;

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

```
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();
    int x = in.readInt();
    int y = in.readInt();
    y = f(x, y);
    out.println(y);
}

public static int f(int m, int n) {
    if (n == 0)
        return 0;
    else
        return m + f(m, n - 1);
}
}
```

- 10.14 Indicate l'output prodotto dall'esecuzione del metodo main della seguente classe qualora si fornisca come dato il numero 7:

```
import prog.io.*;

public class C {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();
        int x = in.readInt();
        int y = f(x, x);
        out.println(y);
    }

    public static int f(int m, int n) {
        if (n + m <= 4)
            return n;
        else if (n > m)
            return 2 + f(m + 1, n - 2);
        else
            return 3 + f(n + 1, m - 2);
    }
}
```

10.4 Metodi con un numero variabile di argomenti

Concludiamo questo capitolo presentando il meccanismo dei *varargs*, una caratteristica introdotta a partire dalla versione 5 del linguaggio, che permette di definire metodi che prevedano un numero variabile di argomenti.

Supponiamo come esempio di voler aggiungere alla classe *Frazione* che abbiamo descritto nel dettaglio nel Capitolo 7, un ulteriore metodo *piu* che permetta di sommare alla frazione che esegue il metodo un numero arbitrario di frazioni fornite come argomento. Un modo per realizzare tale metodo è il seguente:

```
public Frazione piu(Frazione[] args) {
    Frazione somma = this;

    for (Frazione f : args)
        somma = somma.piu(f);

    return somma;
}
```

Cioè il metodo riceve come argomento un array di frazioni e restituisce il riferimento alla frazione ottenuta sommando a quella che esegue il metodo, le frazioni memorizzate nell'array.

Nell'invocazione di questo metodo dobbiamo fornire come argomento l'array delle frazioni che vogliamo sommare alla frazione che esegue il metodo. Ad esempio, dopo l'esecuzione delle istruzioni:

```
Frazione f1 = new Frazione(1,4);
Frazione[] a = {new Frazione(1,8), new Frazione(1,8),
                new Frazione(1,4)};

f1 = f1.piu(a);
```

f1 contiene il riferimento alla frazione $\frac{3}{4}$.

Il meccanismo dei *varargs* permette di utilizzare un metodo analogo a quello appena definito in modo più agevole, evitando di dover inserire esplicitamente in un array gli argomenti da passare al metodo. Ad esempio il metodo precedente può essere riscritto come:

```
public Frazione piu(Frazione... args) {
    Frazione somma = this;

    for (Frazione f : args)
        somma = somma.piu(f);

    return somma;
}
```

I tre punti che seguono il tipo sono la notazione usata per indicare che il metodo prevede un numero variabile di argomenti. Cioè indica al compilatore che il numero di argomenti di tipo **Frazione** che possono essere indicati al momento della chiamata è variabile. Si osservi che il corpo del metodo è identico a quello definito in precedenza. Infatti, `args` all'interno del corpo del metodo è un riferimento ad un array il cui tipo base è indicato prima dei tre punti; quindi nel caso del metodo `piu` è un riferimento di tipo `Frazione[]`.

Quello che cambia in modo sostanziale è il modo in cui possiamo invocare il metodo; ad esempio, le seguenti sono tutte invocazioni ammissibili del metodo `piu`:

```
f1.piu()  
f1.piu(new Frazione(1,8), new Frazione(1,8))  
f1.piu(new Frazione(1,8), new Frazione(1,4), new Frazione(1,4))
```

Al momento dell'esecuzione `args` farà riferimento all'array di tipo `Frazione[]` che contiene, nell'ordine di occorrenza, le frazioni specificate come argomento del metodo. Si osservi che nel primo caso, si tratterà di un riferimento ad un array vuoto, cioè costituito da zero elementi.

Nella dichiarazione di un metodo con un numero variabile di argomenti è possibile utilizzare anche altri argomenti, il solo vincolo è che il meccanismo dei varargs può essere utilizzato unicamente per l'ultimo argomento indicato nella segnatura. A titolo di esempio consideriamo il metodo

```
void printf(String formato, Object... args)
```

della classe `ConsoleOutputManager`. Tale metodo prevede un argomento di tipo `String` (che deve essere obbligatoriamente indicato all'atto della chiamata) e un numero variabile di argomenti di tipo `Object`. Come discuteremo in dettaglio nel Capitolo 13 questo metodo permette di stampare a video gli oggetti indicati formattandoli secondo le direttive specificate dalla stringa `formato`.

Esercizi

- 10.15 Scrivete un metodo `piu` da inserire nella classe `Importo` che, ricevendo un numero variabile di oggetti di tipo `Importo` come argomenti, restituisca l'importo che ne rappresenta la somma.

Eccezioni

Durante l'esecuzione delle applicazioni si possono verificare eventi anomali che, in taluni casi, provocano l'interruzione dell'esecuzione da parte della Java Virtual Machine.

Nella terminologia Java, gli eventi anomali vengono chiamati *eccezioni*. Il linguaggio fornisce la possibilità di trattare questi eventi in maniera opportuna all'interno delle applicazioni mediante il meccanismo di intercettazione delle eccezioni. Nella prima parte di questo capitolo presenteremo alcuni esempi di eccezioni e mostreremo come trattarle. Successivamente indicheremo come sia possibile definire nuove eccezioni e *sollevare* esplicitamente eccezioni all'interno del codice.

11.1 Le eccezioni

Introduciamo l'argomento presentando una semplice anomalia che può verificarsi durante l'esecuzione di un'applicazione: una divisione intera per zero. A tale scopo consideriamo questa applicazione per il calcolo del quoziente di due numeri interi:

```
import prog.io.*;  
  
class Divisione {  
    public static void main(String[] args) {  
        ConsoleInputManager in = new ConsoleInputManager();  
        ConsoleOutputManager out = new ConsoleOutputManager();  
  
        int dividendo = in.readInt("Inserisci il dividendo ");  
        int divisore = in.readInt("Inserisci il divisore ");  
        int quoziente = calcolaQuoziente(dividendo, divisore);  
  
        out.println("Il quoziente è " + quoziente);  
    }  
}
```

```

private static int calcolaQuoziente(int x, int y) {
    return x / y;
}
}

```

All'interno del metodo `main` vengono letti due numeri interi, e mediante la chiamata al metodo `calcolaQuoziente` ne viene calcolato il quoziente e visualizzato il risultato. L'applicazione viene compilata correttamente, e dunque può essere eseguita. Nel caso venga fornito zero come valore del divisore, la Java Virtual Machine non è tuttavia in grado di calcolare il valore dell'espressione `x / y` all'interno del metodo `calcolaQuoziente`; l'esecuzione viene interrotta segnalando l'evento anomalo, cioè l'eccezione che si è verificata:

```

Inserisci il dividendo 8
Inserisci il divisore 0
Exception in thread "main" java.lang.ArithmetricException: / by zero
    at Divisione.calcolaQuoziente(Divisione.java:16)
    at Divisione.main(Divisione.java:10)

```

Il messaggio indica che nel corso dell'esecuzione si è verificata l'eccezione descritta dalla classe `java.lang.ArithmetricException`, precisando che si tratta di una divisione per zero. Vengono inoltre fornite informazioni dettagliate relative al punto del codice in cui ha avuto luogo l'evento: in questo caso nel metodo `Divisione.calcolaQuoziente` (cioè nel metodo `calcolaQuoziente` della classe `Divisione`), alla riga 16 del file `Divisione.java`, richiamato dal metodo `Divisione.main` alla riga 10 del file `Divisione.java`.

Vediamo di comprendere meglio quanto accade al verificarsi dell'evento anomalo. Per mandare in esecuzione un'applicazione, la Java Virtual Machine ne richiama il metodo `main`. In questo caso il metodo `main`, dopo avere effettuato la lettura dei dati, richiama il metodo `calcolaQuoziente`, che restituisce il risultato della divisione. Quando `calcolaQuoziente` termina il proprio compito, il controllo viene ripreso da `main`, che richiederà all'oggetto `out` la visualizzazione del risultato. Questa sequenza di operazioni, in questo caso facilmente desumibile dalla lettura del codice, rappresenta il flusso “normale” di esecuzione. In caso di divisione per zero, questo flusso viene alterato. In particolare, quando si tenta di eseguire una divisione per zero, il metodo `calcolaQuoziente` non è in grado di portare a termine il proprio compito. L'anomalia viene rinviata al metodo chiamante, cioè al `main`, nel punto in cui era stata effettuata la chiamata (riga 10 del file `Divisione.java`, come evidenziato dal messaggio d'errore). Anche in questo caso il metodo `main` non è in grado di proseguire il proprio compito, e dunque non fa altro che rinviare l'anomalia a chi l'ha chiamato, cioè alla Java Virtual Machine, che interrompe l'esecuzione fornendo i messaggi di errore indicati nell'esempio.

Nel caso presentato, l'eccezione è stata sollevata da un'istruzione del linguaggio nel corso della valutazione dell'espressione `x / y`; questo comportamento è documentato nel manuale di Java in relazione al comportamento dell'operatore di divisione intera.¹ Ovviamente, se il valore

¹ Il comportamento dell'operatore di divisione in virgola mobile è diverso. Sia nel tipo `double` sia nel tipo `float` il linguaggio prevede dei valori che rappresentano l'infinito (positivo o negativo) e il valore “not a number” (indicato con

di divisore è diverso da zero, la valutazione dell'espressione va a buon fine e l'esecuzione termina correttamente.

Presentiamo ora un esempio di eccezione sollevata da un metodo di una delle classi della libreria anziché da un'istruzione del linguaggio. Consideriamo il metodo

```
public char charAt(int index)
```

della classe `String`. Questo metodo restituisce il carattere che, nella stringa che lo esegue, si trova nella posizione specificata tramite l'argomento. Che cosa succede se la posizione non esiste? Come descritto nella documentazione del metodo, i valori ammessi per l'argomento sono quelli compresi tra zero e la lunghezza della stringa meno 1, cioè quelli che corrispondono alla convenzione fissata per numerare le posizioni in una stringa. Nel caso venga fornito un valore al di fuori di questo intervallo, il metodo solleva un'eccezione denominata `StringIndexOutOfBoundsException`.

Ecco un'applicazione che illustra questa situazione:

```
import prog.io.*;

class PrelevaCarattere {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        String s = in.readLine("Inserisci una stringa ");
        int indice = in.readInt("Inserisci una posizione ");
        char x = s.charAt(indice);

        out.println("Il carattere in posizione " + indice +
                   " della stringa " + s + " è " + x);
    }
}
```

L'applicazione legge una stringa e un indice, e visualizza il carattere della stringa che si trova nella posizione specificata dall'indice. Quest'applicazione viene correttamente compilata e può essere eseguita. Fornendo come argomenti `pippo` e `7` si ottiene quanto segue:

```
Inserisci una stringa pippo
Inserisci una posizione 7
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 7
        at java.lang.String.charAt(String.java:687)
        at PrelevaCarattere.main(PrelevaCarattere.java:10)
```

`NaN`). Il risultato della divisione per zero di un valore diverso da zero è infinito; il risultato della divisione di zero per zero è `NaN`. Nelle classi involucro sono definite alcune costanti corrispondenti a questi valori.

Anche in questo caso l'esecuzione dell'applicazione è interrotta; la Java Virtual Machine fornisce un resoconto preciso di quanto è avvenuto. In questa circostanza l'eccezione sollevata è `java.lang.StringIndexOutOfBoundsException`.

Anche le eccezioni sono oggetti: sollevando un'eccezione, la Java Virtual Machine crea un'istanza della classe che descrive l'evento verificatosi. Nei messaggi d'errore il nome indicato per l'eccezione è il nome di questa classe. Nell'ultimo esempio l'eccezione è dunque descritta da un'istanza della classe `java.lang.StringIndexOutOfBoundsException` (vedremo più avanti come si possa utilizzare questa istanza intercettando l'eccezione).

Nell'ultimo esempio, a differenza di quanto accadeva nel primo, l'eccezione non viene sollevata direttamente nel metodo `main` dell'applicazione, ma nel metodo `charAt` della classe `String` (per la precisione alla linea 687 del file `String.java`): l'eccezione viene quindi propagata al metodo `main`, alla riga 10 del file `PrelevaCarattere.java`, cioè alla riga contenente l'invocazione del metodo `charAt`.

Le eccezioni sollevate nei due esempi precedenti sono eccezioni predefinite del linguaggio Java. Tutte le eccezioni sono sottoclassi della classe `Throwable` ("sollevabile"). Nella Figura 11.1 è illustrata la collocazione delle eccezioni `ArithmetricException` e `StringIndexOutOfBoundsException` all'interno della gerarchia delle eccezioni di Java.

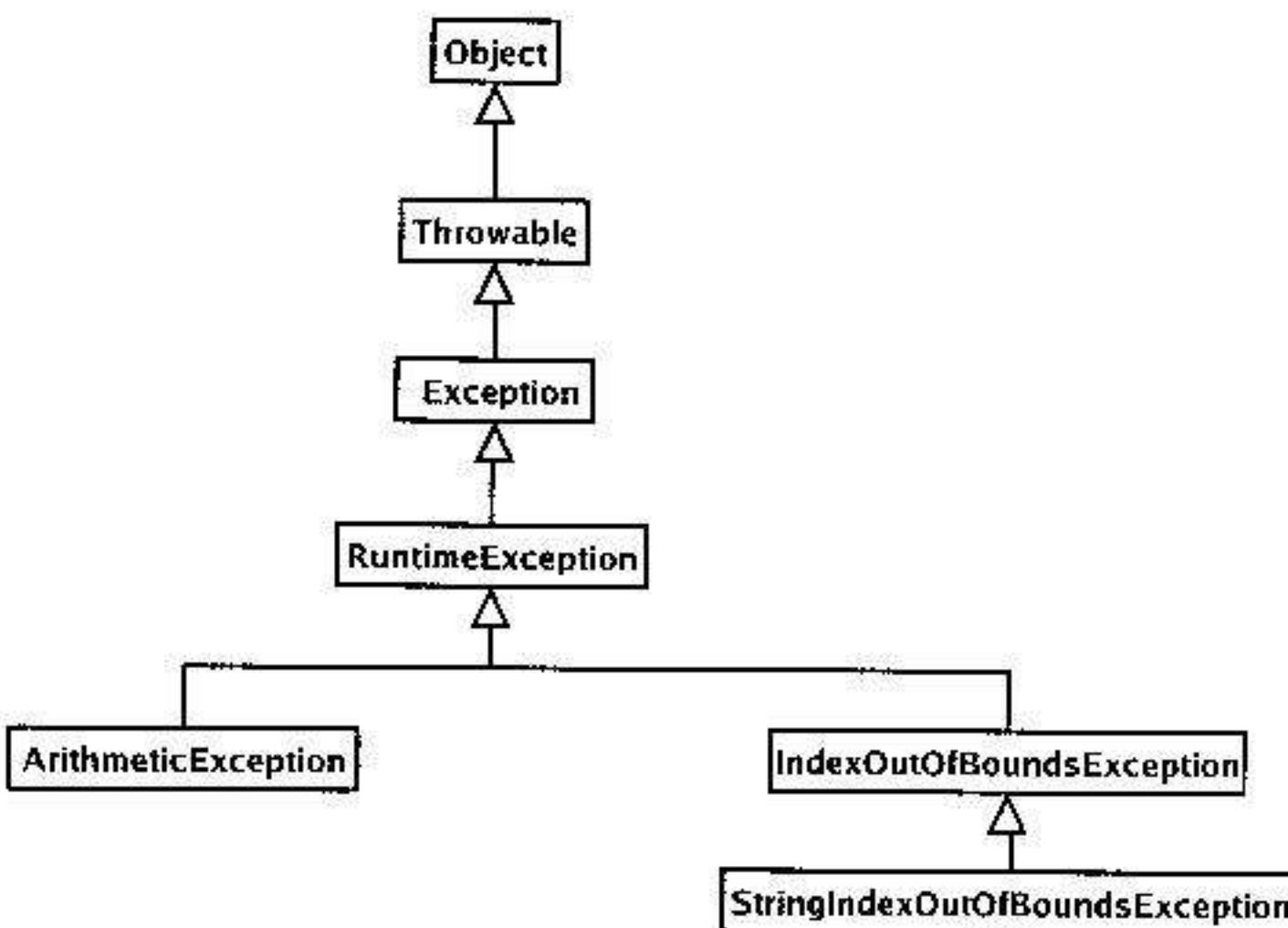


Figura 11.1 Un frammento della gerarchia delle eccezioni di Java.

Per scrivere programmi corretti è ovviamente necessario sapere quali sono le condizioni eccezionali che si potrebbero verificare durante l'esecuzione di un metodo o di un costruttore. Pertanto la documentazione delle classi deve fornire, insieme con la descrizione dei metodi e dei costruttori, l'indicazione delle eccezioni che essi possono sollevare.

In ambedue gli esempi presentati si può riscrivere il codice in modo che la situazione anomala, cioè l'eccezione, venga evitata, inserendo controlli preventivi sui valori coinvolti nell'i-

struzione che potrebbe sollevare l'eccezione. Nel secondo esempio potremmo prevenire l'errore inserendo le ultime due istruzioni del metodo `main` in una selezione in cui controlliamo che l'indice considerato sia valido:

```
if (indice >= 0 && indice < s.length()) {
    char x = s.charAt(indice);
    out.println("Il carattere in posizione " + indice +
                " della stringa " + s + " è " + x);
} else
    out.println("Non esiste alcun carattere in posizione " + indice);
```

Nell'esempio presentato, o in un esempio come il precedente, relativo alla divisione, questo è il modo più naturale per gestire le situazioni anomale. In altri casi, tuttavia, l'inserimento di controlli di questo genere non rappresenta la soluzione migliore, in quanto il punto in cui l'eccezione potrebbe essere sollevata (e dunque prevenuta) è lontano dal punto in cui intervenire per porvi rimedio. In questo caso l'inserimento di controlli può appesantire notevolmente il codice, rendendolo illeggibile. Presenteremo un esempio di questo tipo più avanti nel capitolo, quando descriveremo un'applicazione per la valutazione di espressioni scritte in notazione postfissa.

11.2 Come intercettare le eccezioni: l'istruzione try-catch

È possibile gestire gli eventi anomali, una volta che si sono verificati, con apposite azioni in un punto opportuno del codice mediante il meccanismo di *intervettazione delle eccezioni*.

Riprendiamo l'esempio dell'applicazione `PrelevaCarattere`. Anziché prevenire l'eccezione che il metodo `charAt` può sollevare, potremmo decidere di procedere come segue:²

- si tenta di eseguire l'operazione desiderata, cioè di ottenere tramite la chiamata `s.charAt(indice)` il carattere in posizione `indice` della stringa `s`;
- se l'operazione non va a buon fine, si rimedia eseguendo un'azione alternativa.

Dire che “l'operazione non va a buon fine” equivale a dire che viene *solllevata* l'eccezione: normalmente, come evidenziato nell'esempio precedente, ciò provoca la terminazione anomala dell'esecuzione. Tuttavia, se si *intercetta* l'eccezione, si può rimediare senza interrompere l'esecuzione, intraprendendo azioni alternative.

Per *tentare* l'esecuzione di un blocco di codice e *intervettare* eventuali eccezioni, Java fornisce l'istruzione `try-catch`, che ha il seguente schema:

```
try {
    // Codice che potrebbe generare eccezioni
    blocco_try
```

² L'esempio ha come unico scopo quello di mostrare il costrutto `try-catch` che permette di intercettare le eccezioni. Ribadiamo che in questo caso è molto meglio prevenire l'eccezione utilizzando un controllo sulla variabile `indice`, come indicato sopra.

```

}
catch (Tipo_Eccezione1 id1) {
    // Gestisce le eccezioni di Tipo_Eccezione1
    blocco_gestore1
} catch (Tipo_Eccezione2 id2) {
    // Gestisce le eccezioni di Tipo_Eccezione2
    blocco_gestore2
}
...

```

Il codice in *blocco_try* è il blocco di codice che si vorrebbe eseguire e che può sollevare eccezioni; ciascuna clausola *catch* costituisce invece il gestore di un determinato tipo di eccezione, il tipo specificato come argomento della clausola. Ad esempio nello schema descritto sopra il primo blocco *catch* è preposto a gestire le eccezioni di tipo *Tipo_Eccezione1*; l'identificatore *id1* specificato dopo il tipo dell'eccezione può essere utilizzato per fare riferimento all'oggetto eccezione intercettato dal gestore all'interno del codice *blocco_gestore1*. Non esistono limiti al numero di gestori che possono essere presenti dopo un blocco *try*.

Quando esegue un blocco *try-catch*, la Java Virtual Machine si comporta nel modo seguente.

- (1) Inizia a eseguire il codice presente nel blocco *try*; se durante l'esecuzione di questo codice non viene sollevata alcuna eccezione, l'esecuzione prosegue con la prima istruzione successiva al blocco *try-catch*.
- (2) Se durante l'esecuzione del codice nel blocco *try* viene sollevata un'eccezione, la Java Virtual Machine scorre la lista degli argomenti dei blocchi *catch* nell'ordine in cui sono scritti, alla ricerca di un tipo cui possa essere ricondotto quello dell'eccezione sollevata (cioè che sia uguale o supertipo di quello dell'eccezione sollevata). Quando lo trova, esegue il codice del corrispondente blocco *catch*, terminato il quale passa a eseguire la prima istruzione che segue il blocco *try-catch*.
- (3) Se nessuno dei blocchi *catch* è preposto a intercettare l'eccezione sollevata, la Java Virtual Machine continua l'esecuzione esattamente come se il blocco *try-catch* non fosse presente. In particolare se ciò avviene durante l'esecuzione di un metodo *main*, come nei casi che stiamo considerando, questo implica l'interruzione dell'esecuzione e la segnalazione dell'anomalia all'utente.

Ecco l'applicazione *PrelevaCarattere*, riscritta utilizzando il costrutto *try-catch*:

```

import prog.io.*;

class PrelevaCarattere {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

```

```

String s = in.readLine("Inserisci una stringa ");
int indice = in.readInt("Inserisci una posizione ");
try {
    char x = s.charAt(indice);

    out.println("Il carattere in posizione " + indice +
               " della stringa " + s + " è " + x);
} catch (StringIndexOutOfBoundsException e) {
    out.println("Non esiste alcun carattere in posizione " + indice);
}
}
}

```

Ecco un esempio di esecuzione:

```

Inserisci una stringa pippo
Inserisci una posizione 7
Non esiste alcun carattere in posizione 7

```

Si osservi che, durante l'esecuzione del metodo `charAt`, è stata sollevata in questo caso una `StringIndexOutOfBoundsException`. Dunque, come descritto in precedenza, la Java Virtual Machine è passata a eseguire il blocco `catch` corrispondente a questa eccezione (in questo caso l'unico blocco `catch` presente), in cui viene visualizzato il messaggio di errore.

Se nel blocco `catch` avessimo indicato come tipo `Exception` o `RuntimeException`, l'effetto sarebbe stato il medesimo, perché entrambe sono superclassi di `StringIndexOutOfBoundsException`. Se avessimo invece indicato `ArithmetricException`, in caso di errore l'esecuzione sarebbe stata interrotta (come nella versione iniziale priva di trattamento delle eccezioni) per la mancanza del gestore appropriato.

Nell'intestazione del blocco `catch`, dopo il tipo dell'eccezione viene scritto un identificatore. Questa è una vera e propria dichiarazione: nell'esempio precedente abbiamo dichiarato un identificatore di nome `e` di tipo `StringIndexOutOfBoundsException`. Questo identificatore può essere utilizzato all'interno del blocco `catch` per riferirsi all'oggetto che rappresenta l'eccezione. In altre parole:

- quando è sollevata un'eccezione, viene creato un oggetto che descrive quanto è accaduto;
- nel blocco `catch` che gestisce l'eccezione è possibile utilizzare tale oggetto, ad esempio invocandone i metodi.

Nelle classi corrispondenti alle eccezioni viene di solito fornito un metodo `toString` che restituisce una stringa che descrive l'errore. Proviamo a riscrivere l'applicazione `PrelevaCarattere` visualizzando in caso di errore il risultato del metodo `toString` dell'oggetto eccezione:

```
import prog.io.*;
```

```

class PrelevaCarattere {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        String s = in.readLine("Inserisci una stringa ");
        int indice = in.readInt("Inserisci una posizione ");
        try {
            char x = s.charAt(indice);

            out.println("Il carattere in posizione " + indice +
                       " della stringa " + s + " è " + x);
        } catch (StringIndexOutOfBoundsException e) {
            out.println(e.toString());
        }
    }
}

```

Ecco di nuovo un esempio di esecuzione:

```

Inserisci una stringa pippo
Inserisci una posizione 7
java.lang.StringIndexOutOfBoundsException: String index out of range: 7

```

Si osservi che, mentre un messaggio di questo tipo potrebbe essere utile al programmatore, all’utente finale risulta poco comprensibile.

11.3 Rientro dai metodi ed eccezioni

Normalmente, quando l’esecuzione di un metodo termina, viene ripresa l’esecuzione del metodo chiamante. Ad esempio, se un metodo `main` ha chiamato un metodo `a`, che a sua volta ha chiamato `b`, che a sua volta ha chiamato `c`, al termine dell’esecuzione di `c` il controllo viene ripreso da `b`; al termine dell’esecuzione di `b` il controllo viene ripreso da `a` e, infine, al termine dell’esecuzione di `a` il controllo viene ripreso da `main`.

Abbiamo visto che alle operazioni di chiamata e di rientro dai metodi corrispondono operazioni all’interno della memoria della Java Virtual Machine, in particolare nell’area denominata stack. Quando viene invocato un metodo, sullo stack si crea un record di attivazione per i dati del metodo. Tale record viene distrutto al momento del rientro dal metodo.

Qualora venga sollevata un’eccezione, le cose possono andare in maniera diversa. Supponiamo che durante l’esecuzione di `c` venga sollevata un’eccezione. Se il metodo `c` è in grado di intercettarla, allora esso, dopo avere risolto il problema, termina la propria esecuzione effettuando le normali operazioni di rientro, e il controllo viene ripreso da `b`.

Se invece `c` non intercetta l'eccezione, la sua esecuzione viene interrotta e l'eccezione viene riportata al metodo chiamante, cioè `a`. In particolare, nello stack sarà distrutto il record di attivazione di `c` e l'eccezione verrà sollevata nel punto in cui il metodo `b` aveva invocato `c`. Anche in questo caso ci sono due possibilità: `b` intercetta l'eccezione e termina poi normalmente la propria esecuzione restituendo il controllo ad `a`, oppure `b` non intercetta l'eccezione, la sua esecuzione viene interrotta e l'eccezione viene risollevata in `a`.

In sostanza, un'eccezione sollevata in un metodo o in un costruttore viene propagata nella catena dei chiamanti finché non si trova un metodo o un costruttore che la intercetti, o fino a provocare l'interruzione dell'esecuzione (se nemmeno `main` è in grado di intercettarla). Quando un metodo o un costruttore rinvia l'eccezione al chiamante, la sua esecuzione viene interrotta e il suo record di attivazione è eliminato dallo stack.

Esercizi

- 11.1 Riscrivete il metodo `main` della classe `Divisione` in modo che intercetti l'eccezione sollevata dal metodo `calcolaQuoziente`.

11.4 Esempio: una calcolatrice in notazione postfissa

La notazione comunemente utilizzata per scrivere le espressioni aritmetiche, in cui il simbolo di operazione viene scritto tra gli operandi ai quali si applica, è chiamata *notazione infissa*. Il calcolo di espressioni in notazione infissa è complicato, in quanto occorre tenere conto delle regole di precedenza degli operatori (ad esempio del fatto che le moltiplicazioni e le divisioni precedono le somme e le sottrazioni) e della presenza di eventuali parentesi. Ad esempio, leggendo da sinistra a destra l'espressione

$$12 + (8 - 2) * (5 - 3)$$

si ottiene, prima di tutto, il valore 12, seguito dal simbolo di addizione. In realtà l'addizione sarà l'ultima operazione a essere applicata; prima occorre calcolare il risultato della moltiplicazione, che a sua volta richiede il calcolo delle due sottrazioni. Dal punto di vista del calcolo risulta più semplice una notazione differente, in cui prima si specificano gli operandi e poi l'operazione da applicare. Tale notazione viene detta *notazione postfissa*. Ad esempio l'espressione precedente può essere scritta in notazione postfissa come:

$$12\ 8\ 2\ -\ 5\ 3\ -\ *\ +$$

Un'espressione in notazione postfissa viene valutata applicando ogni operazione ai due valori o ai risultati di altre operazioni che la precedono. Nell'esempio i due operatori `-` si applicano, rispettivamente, a 8 e 2, e a 5 e 3; l'operatore `*` si applica ai risultati delle due sottrazioni; l'operatore `+` si applica al valore 12 e al risultato della moltiplicazione.

Per valutare l'espressione potremmo anche fornire la seguente regola: ogni volta che un simbolo di operazione è preceduto da due valori si può semplificare l'espressione sostituendo i

due valori e il simbolo di operazione con il risultato. Ad esempio l'espressione scritta sopra può essere semplificata calcolando le sottrazioni:

$$12 \ 6 \ 2 \ * \ +$$

A questo punto si può calcolare il prodotto ottenendo:

$$12 \ 12 \ +$$

Infine, con una nuova semplificazione, si ottiene il risultato, cioè 24.

Nella notazione postfissa, oltre a non esserci regole di precedenza tra gli operatori, non è necessario usare parentesi. Ad esempio l'espressione infissa precedente diventa senza parentesi:

$$12 + 8 - 2 * 5 - 3$$

La sua rappresentazione in forma postfissa è:

$$12 \ 8 \ + \ 2 \ 5 \ * \ - \ 3 \ -$$

Il valore delle espressioni in notazione postfissa può essere calcolato facilmente servendosi di una struttura a *pila* (o *stack*). Ricordiamo che una pila è una struttura in cui è possibile aggiungere o eliminare elementi solo in cima a essa. Le operazioni di aggiunta e di eliminazione dell'elemento in cima alla pila sono denominate, rispettivamente, *push* e *pop*. Una struttura a pila è detta anche struttura *Last In First Out*, in quanto il primo elemento che può essere eliminato è l'ultimo inserito nella struttura.

Per calcolare il valore di espressioni in notazione postfissa utilizziamo una pila di numeri, inizialmente vuota. L'algoritmo per il calcolo non fa altro che scandire l'espressione, da sinistra verso destra, applicando le seguenti regole:

- quando si legge un numero, lo si inserisce in cima alla pila;
- quando si legge un simbolo di operazione, si prelevano i due numeri che si trovano più in alto sulla pila, si applica loro l'operazione e si inserisce il risultato sulla pila.

Se l'espressione è formata correttamente, alla fine di questo procedimento la pila conterrà, come unico elemento, il risultato.

Illustriamo i passi per il calcolo dell'espressione $8 \ 1 \ - \ 4 \ 3 \ 5 \ * \ + \ *$.

- Inizialmente la pila è vuota:

<i>pila</i>	<i>parte di espressione che resta da leggere</i>
	$8 \ 1 \ - \ 4 \ 3 \ 5 \ * \ + \ *$

- Lettura del numero 8. Il numero viene inserito in cima alla pila:

<i>pila</i>	<i>parte di espressione che resta da leggere</i>
8	$1 \ - \ 4 \ 3 \ 5 \ * \ + \ *$

- Lettura del numero 1. Il numero viene inserito in cima alla pila:

<i>pila</i>	<i>parte di espressione che resta da leggere</i>
1	— 4 3 5 * + *
8	

- Lettura del segno —. Si prelevano i due valori più in alto nella pila, cioè 1 e 8 (la pila quindi resta vuota); si applica l'operazione ai due numeri (il primo operando è l'ultimo numero prelevato, cioè 8). Si inserisce il risultato dell'operazione in cima alla pila:

<i>pila</i>	<i>parte di espressione che resta da leggere</i>
7	4 3 5 * + *

- Lettura del numero 4:

<i>pila</i>	<i>parte di espressione che resta da leggere</i>
4	3 5 * + *
7	

- Lettura del numero 3:

<i>pila</i>	<i>parte di espressione che resta da leggere</i>
3	5 * + *
4	
7	

- Lettura del numero 5:

<i>pila</i>	<i>parte di espressione che resta da leggere</i>
5	* + *
3	
4	
7	

- Lettura del segno *:

<i>pila</i>	<i>parte di espressione che resta da leggere</i>
15	+ *
4	
7	

- Lettura del segno +:

<i>pila</i>	<i>parte di espressione che resta da leggere</i>
19	*
7	

- Lettura del segno *:

pila parte di espressione che resta da leggere
133

Esercizi

11.2 Riscrivete le seguenti espressioni utilizzando la notazione postfissa.

- (1) $3 + 5 * (20 - 4 * 3 + 5) / (10 + 2 * 3)$
- (2) $3 * 3 + (4 + 6 - (2 + 3) * (5 - 10)) - (7 + 2)$

Descrivete poi i passi effettuati per calcolare i risultati delle espressioni tramite una pila.

11.3 Riscrivete in notazione infissa ognuna delle seguenti espressioni postfisse.

- (1) $10\ 4\ 3\ *\ 2\ /\ +\ 5\ +\ 3\ 6\ 8\ +\ 4\ *\ +\ -$
- (2) $10\ 4\ +\ 3\ 2\ /\ 5\ 3\ -\ +\ *\ 6\ 8\ +\ 4\ *\ +$

Implementazione della calcolatrice

Vogliamo ora scrivere un'applicazione in grado di calcolare il valore di espressioni scritte in notazione postfissa. Per semplicità supporremo di ricevere gli elementi che costituiscono l'espressione su righe separate (per la lettura dell'espressione scritta su un'unica riga si vedano gli esercizi successivi): una riga può contenere un numero intero privo di segno oppure uno dei quattro simboli di operazione: +, -, * o /. La fine dell'espressione sarà indicata da una riga contenente il solo carattere =.

Per scrivere il codice risulta utile la classe generica `Stack` (sottoclasse di `Vector` definita nel package `java.util`). La classe ha un tipo parametro `E`; le sue istanze rappresentano pile di oggetti del tipo argomento corrispondente a `E`. La classe dispone di:

- un costruttore privo di argomenti che crea una pila vuota;
- un metodo `push(E o)` che aggiunge in cima alla pila che esegue il metodo l'oggetto di cui riceve il riferimento tramite il parametro;³
- un metodo `E pop()` che toglie dalla pila che esegue il metodo l'elemento che si trova più in alto restituendone il riferimento;
- un metodo `boolean empty()` che restituisce `true` se e solo se la pila che esegue il metodo è vuota.

In base a quanto appena stabilito e in base all'algoritmo per la valutazione descritto nel paragrafo precedente, la codifica del metodo `main` dell'applicazione può essere basata su questo schema:

³ Il metodo restituisce un riferimento di tipo `E` all'oggetto inserito nella pila; tuttavia ignoreremo quanto viene restituito, utilizzando il metodo come se restituisse `void`.

```

crea una pila vuota
leggi una riga
while (ciò che hai letto è diverso da =) {
    elabora ciò che hai letto
    leggi una riga
}
preleva il risultato dalla cima della pila
comunica il risultato

```

Traduciamo questo schema in linguaggio Java. Come mostrato nell'esempio precedente, la pila è destinata a contenere numeri interi, corrispondenti ai dati o ai risultati delle operazioni. Pertanto per rappresentarla utilizziamo un'istanza della classe Stack, con tipo argomento Integer. Memorizzeremo il riferimento a essa in una variabile di nome `pila`. Iniziamo a costruire la pila usando il costruttore privo di argomenti della classe Stack; l'oggetto ottenuto in questo modo rappresenta una pila vuota. Esamineremo più avanti la fase di elaborazione.

Al termine della fase di elaborazione, cioè dopo il ciclo while, dobbiamo prelevare l'elemento che si trova in cima alla pila, che rappresenta il risultato dell'espressione. A tale scopo possiamo chiedere all'oggetto `pila` di eseguire il proprio metodo `pop` scrivendo l'espressione:

```
pila.pop()
```

Ecco lo schema generale codificato in Java:

```

Stack<Integer> pila = new Stack<Integer>();

out.println("Inserire l'espressione da esaminare");
out.println("(un elemento per riga, = per terminare)");
String riga = in.readLine();
while (!riga.equals("=")) {

    //elaborazione della riga letta
    ...

    riga = in.readLine();
}

Integer risultato = pila.pop();
out.println("Il risultato è " + risultato);

```

Sviluppiamo ora la fase di elaborazione della stringa. Supponendo che le espressioni siano costruite correttamente è necessario considerare i seguenti casi.

- Se è stato letto un simbolo di operazione, occorre prelevare dalla pila i due operandi, calcolare il risultato dell'operazione e porlo in cima alla pila. Per prelevare i due operandi dalla pila faremo ricorso al metodo `pop` (si osservi che il primo operando prelevato dalla pila è quello di destra):

```
Integer dx = pila.pop();
Integer sx = pila.pop();
```

Una volta ottenuti i due operandi, possiamo calcolare il risultato. Per comodità rimandiamo questo compito a un metodo denominato `calcola`, collocato sempre nella stessa classe, che riceverà come argomenti i due operandi interi e il segno dell'operazione di tipo `char`, e restituirà il risultato dell'operazione:

```
Integer risultato = calcola(sx, dx, segno dell'operazione);
```

Il metodo `calcola` sarà sviluppato poco più avanti. L'oggetto ottenuto come risultato dovrà infine essere caricato sulla pila. Quest'ultima operazione viene svolta dalla pila stessa mediante il metodo `push`:

```
pila.push(risultato);
```

- Se è stato letto un operando, è necessario portarlo in cima alla pila. A tale scopo costruiamo un oggetto `Integer`, corrispondente alla stringa letta, che dovrà essere caricato in cima alla pila:

```
Integer numero = new Integer(riga);
pila.push(numero);
```

Possiamo basare la codifica di questa fase su un costrutto `switch`, in cui il selettore è il primo carattere della stringa letta (per ora supponiamo che l'input sia fornito nel formato corretto senza considerare situazioni di errore):

```
Integer sx, dx, risultato;
char selettore = riga.charAt(0);
switch (selettore) {
    case '+':
    case '-':
    case '*':
    case '/':
        dx = pila.pop();
        sx = pila.pop();
        risultato = calcola(sx, dx, selettore);
        pila.push(risultato);
        break;
    default:
        //se non è un simbolo di operazione, dev'essere un numero
        Integer numero = new Integer(riga);
        pila.push(numero);
        break;
}
```

Scriviamo ora il codice del metodo `calcola`. Osserviamo, prima di tutto, che il solo scopo del metodo è quello di svolgere una parte del compito del metodo `main`: come quest'ultimo, non è legato a un particolare oggetto. Perciò definiamo il metodo come statico. Inoltre il suo compito è legato esclusivamente all'implementazione di `main`, che stiamo sviluppando; un'implementazione diversa potrebbe non servirsi di questo metodo. Decidiamo quindi di definirlo come privato.

Ricordiamo che il metodo deve ricevere due numeri interi e un carattere, che rappresenta un simbolo di operazione, e deve restituire il risultato intero dell'operazione applicata ai due numeri. La codifica è basata su un costrutto `switch` grazie al quale, in base al carattere ricevuto tramite il parametro, viene selezionata l'operazione da eseguire:⁴

```
private static int calcola(int opSx, int opDx, char segno) {
    int ris = 0;
    switch (segno) {
        case '+':
            ris = opSx + opDx;
            break;
        case '-':
            ris = opSx - opDx;
            break;
        case '*':
            ris = opSx * opDx;
            break;
        case '/':
            ris = opSx / opDx;
            break;
    }
    return ris;
}
```

Si osservi l'intestazione del metodo: i primi due parametri e il risultato sono del tipo primitivo `int` e non del tipo riferimento `Integer`. Questa scelta è coerente con l'elaborazione che deve fare il metodo, che riguarda i valori interi ed è basata sulle operazioni del tipo primitivo. D'altra parte, nell'invocazione di `calcola` in `main` i primi due argomenti sono di tipo `Integer`, il risultato prodotto dal metodo viene assegnato a una variabile di tipo `Integer`. Quest'uso è possibile grazie ai meccanismi di unboxing e autoboxing dei tipi primitivi presentati nel Capitolo 4. In particolare, al momento dell'invocazione del metodo `calcola` verrà effettuato l'unboxing

⁴ Per come è costruita l'applicazione, quando il metodo sarà chiamato, l'argomento `segno` conterrà sicuramente uno dei quattro simboli di operazione: dunque nell'istruzione `switch` verrà assegnato senz'altro un valore alla variabile `ris`. Pertanto l'inizializzazione di `ris` a zero è, dal punto di vista logico, superflua. Tuttavia il compilatore non conosce la logica dei programmi. In particolare, se non inizializzassimo `ris` a zero, il compilatore stabilirebbe che al termine dell'istruzione `switch` la variabile `ris` potrebbe non essere inizializzata (qualora non venisse eseguito alcuno dei casi elencati). Questa conclusione porterebbe a una segnalazione di errore nella successiva istruzione di restituzione del risultato.

dei primi due argomenti. Dopo il rientro dal metodo, per assegnarne il valore restituito di tipo `int` alla variabile risultato di tipo `Integer`, verrà effettuato l'autoboxing.

Ecco il codice completo della classe Postfissa:

```

import prog.io.*;
import java.util.Stack;
}

class Postfissa {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //crea la pila
        Stack<Integer> pila = new Stack<Integer>();

        out.println("Inserire l'espressione da esaminare");
        out.println("(un elemento per riga, = per terminare)");
        String riga = in.readLine();
        while (!riga.equals("=")) {

            //elaborazione della riga letta
            Integer sx, dx, risultato;
            char selettore = riga.charAt(0);
            switch (selettore) {
                case '+':
                case '-':
                case '*':
                case '/':
                    dx = pila.pop();
                    sx = pila.pop();
                    risultato = calcola(sx, dx, selettore);
                    pila.push(risultato);
                    break;
                default:
                    //se non è un simbolo di operazione, dev'essere un numero
                    Integer numero = new Integer(riga);
                    pila.push(numero);
                    break;
            }
            //lettura di un'altra riga
        }
    }
}

```

```
    riga = in.readLine();
}

//preleva il risultato dalla pila e lo comunica
Integer risultato = pila.pop();
out.println("Il risultato è " + risultato);
}

private static int calcola(int opSx, int opDx, char segno) {
    int ris = 0;
    switch (segno) {
        case '+':
            ris = opSx + opDx;
            break;
        case '-':
            ris = opSx - opDx;
            break;
        case '*':
            ris = opSx * opDx;
            break;
        case '/':
            ris = opSx / opDx;
            break;
    }
    return ris;
}

}
```

Ecco un esempio di esecuzione:

Inserire l'espressione da esaminare
(un elemento per riga, = per terminare)

8
1
-
4
3
5
*
+
*
=

Il risultato è 133

Esercizi

- 11.4 Modificate la classe Postfissa in modo che l'utente possa inserire tutta l'espressione da calcolare in un'unica riga. A tale scopo occorre leggere inizialmente tutta la riga contenente l'espressione suddividendola nei vari elementi (numeri e simboli di operazione) che la costituiscono (si consiglia di utilizzare la classe StringTokenizer). In questo caso non è necessario indicare la fine dell'espressione con il carattere =.
- 11.5 Modificate la classe Postfissa in modo che vengano visualizzati i passi effettuati per calcolare l'espressione e i risultati intermedi ottenuti. Ad esempio, se l'espressione inserita è:

10 5 4 - 3 2 * + -

l'output prodotto dovrà essere:

```
5 - 4 = 1
3 * 2 = 6
1 + 6 = 7
10 - 7 = 3
Il risultato è 3
```

Trattamento delle situazioni anomale

L'applicazione Postfissa è stata sviluppata nell'ipotesi che l'input fornito dall'utente sia in formato corretto e che non si verifichino condizioni eccezionali. Analizziamo ora le situazioni critiche che potrebbero verificarsi in base a ciò che l'utente inserisce. Studieremo poi come porvi rimedio. In particolare evidenziamo questi quattro possibili tipi di problemi.

- (1) L'espressione contiene una divisione per zero. Esempio d'esecuzione:

```
Inserire l'espressione da esaminare
(un elemento per riga, = per terminare)
5
4
+
0
/
Exception in thread "main" java.lang.ArithmetricException: / by zero
at Postfissa.calcola(Postfissa.java:61)
at Postfissa.main(Postfissa.java:29)
```

(2) L'utente inserisce caratteri non corretti. Esempio:

```
Inserire l'espressione da esaminare  
(un elemento per riga, = per terminare)  
2  
3  
+  
2ac  
Exception in thread "main" java.lang.NumberFormatException:  
For input string: "2ac"  
    at java.lang.NumberFormatException.  
        forInputString(NumberFormatException.java:48)  
    at java.lang.Integer.parseInt(Integer.java:456)  
    at java.lang.Integer.<init>(Integer.java:620)  
    at Postfissa.main(Postfissa.java:34)
```

(3) L'espressione contiene un numero insufficiente di operandi. Esempio:

```
Inserire l'espressione da esaminare  
(un elemento per riga, = per terminare)  
5  
+  
Exception in thread "main" java.util.EmptyStackException  
    at java.util.Stack.peek(Stack.java:85)  
    at java.util.Stack.pop(Stack.java:67)  
    at Postfissa.main(Postfissa.java:28)
```

(4) L'espressione contiene troppi operandi. Esempio:

```
Inserire l'espressione da esaminare  
(un elemento per riga, = per terminare)  
6  
3  
4  
+  
=  
Il risultato è 7
```

Nei primi tre casi l'esecuzione viene interrotta. La Java Virtual Machine fornisce un messaggio di errore in merito a quanto si è verificato. Nell'ultimo caso l'esecuzione viene invece terminata fornendo un risultato scorretto, senza evidenziare l'inconsistenza dell'input fornito.⁵

⁵ In realtà vi è un'altra possibile anomalia, nel caso l'utente inserisca una riga vuota. Questo problema è legato al particolare formato, estremamente elementare, con cui vengono inseriti i dati e, per semplicità, non viene considerato in questa trattazione.

Si osservi che i messaggi d'errore contengono il nome dell'eccezione sollevata (ad esempio `java.lang.ArithmetricException`) ed eventualmente informazioni aggiuntive (come / by zero), e una serie di informazioni relative al punto in cui si è verificato l'errore (ad esempio nel primo caso l'eccezione è stata sollevata nel metodo `calcola` della classe `Postfissa`, alla riga 61, ed è stata rinviata al metodo `main` alla riga 29, cioè alla riga in cui si è avuta la chiamata di `calcola`). Questi messaggi possono essere utili al programmatore durante la messa a punto del programma, ma risultano poco comprensibili a un utente che voglia usare il programma senza doverne necessariamente conoscere i dettagli implementativi.

Iniziamo a considerare l'ultima situazione d'errore evidenziata, cioè il caso in cui siano stati forniti troppi operandi. Ripensando all'algoritmo utilizzato per valutare le espressioni possiamo osservare che, per il caso dell'esempio, il risultato fornito 7 è la somma dei due operandi 3 e 4. Sulla pila è rimasto inutilizzato, però, il dato 6. Non è difficile convincersi che, se l'espressione è costruita correttamente, al termine dei calcoli deve restare sulla pila esclusivamente il risultato. Questa situazione di errore può essere trattata facilmente con una selezione: dopo avere prelevato il risultato dalla pila si controlla che questa sia rimasta vuota e, in caso contrario, si fornisce un messaggio d'errore in luogo del risultato. Nel metodo `main` dell'applicazione `Postfissa`, la parte di comunicazione del risultato può essere in sostanza riscritta come:

```
Integer risultato = pila.pop();
if (pila.empty())
    out.println("Il risultato è " + risultato);
else
    out.println("L'espressione contiene troppi operandi");
```

Anche le altre situazioni di errore potrebbero essere trattate utilizzando istruzioni di selezione. Ad esempio, per evitare la divisione per zero si può inserire un controllo sul divisore, o per evitare l'errore dovuto al numero insufficiente di operandi si può, prima di ogni chiamata del metodo `pop`, controllare che la pila non sia vuota. Tuttavia questi controlli finiscono per appesantire notevolmente il codice, nascondendo le parti fondamentali della sua struttura e rendendolo poco leggibile.

A titolo d'esempio consideriamo il problema della divisione per zero. L'operazione di divisione viene effettuata nel metodo `calcola`. Una soluzione potrebbe consistere nell'introdurre, come ulteriore parametro del metodo `calcola`, il riferimento a un apposito oggetto; mediante la modifica dello stato di tale oggetto, il metodo `calcola` può segnalare al chiamante il verificarsi di una situazione anomala. A sua volta il metodo `main`, controllando lo stato di tale oggetto, può verificare se il metodo `calcola` è stato eseguito senza problemi o se si sono verificate anomalie: in questo secondo caso il metodo `main` dovrà interrompere la valutazione dell'espressione e fornire un messaggio d'errore.

Si osservi che l'inserimento di questi controlli richiede profondi cambiamenti nella struttura del metodo `main`: sia l'istruzione `switch` sia il ciclo `while` dovranno essere completamente riorganizzati (un esercizio utile è appunto quello di riscrivere l'applicazione trattando le situazioni di errore con selezioni).

L'uso delle eccezioni permette di risolvere questi problemi in maniera elegante e pulita, creando nel codice una separazione tra ciò che dev'essere eseguito normalmente e il trattamento

delle situazioni anomale. In sostanza, possiamo immaginare due modalità di terminazione di un metodo:

- terminazione “normale”, con l'esecuzione dell'istruzione `return` mediante la quale il metodo restituisce al chiamante il risultato prodotto;
- terminazione “anomala”, con la quale il metodo rinvia al chiamante un'eccezione che sarà risollevata nel punto in cui il metodo è stato chiamato.

In particolare i messaggi d'errore degli esempi precedenti evidenziano i punti in cui le eccezioni sono state sollevate per la prima volta e i punti in cui sono state via via rinviate ai metodi chiamanti.

Mostriremo ora come, utilizzando il trattamento delle eccezioni nella classe `Postfissa`, sia possibile scrivere il codice che tratta la situazione anomala nel punto che il programmatore ritiene più opportuno senza necessità di riempire il resto del codice di numerosi controlli che lo renderebbero più difficile da leggere. Grazie al costrutto `try-catch` possiamo scrivere da una parte (all'interno del blocco `try`) il codice da eseguire nelle situazioni “normali” e, separatamente (all'interno del blocco `catch`), il codice per trattare le situazioni anomale.

Ecco una prima versione in cui trattiamo indistintamente i tre tipi di eccezioni fornendo un messaggio d'errore prima di terminare l'esecuzione dell'applicazione. Si noti che il codice inserito nella parte `try` è identico a quello della versione precedente. La parte `catch` si trova alla fine del metodo `main` in quanto abbiamo scelto, in caso di errore, di fornire un messaggio di errore e terminare l'esecuzione.

```
import prog.io.*;
import java.util.Stack;

class Postfissa {

    public static void main(String[] args) {
        //predisposizione dei canali di comunicazione
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //crea la pila
        Stack<Integer> pila = new Stack<Integer>();

        out.println("Inserire l'espressione da esaminare");
        out.println("(un elemento per riga, = per terminare)");
        String riga = in.readLine();
        try {
            //il codice che segue è esattamente quello della versione
            //precedente
            while (!riga.equals("=")) {
```

```
//elaborazione della riga letta
Integer sx, dx, risultato;
char selettore = riga.charAt(0);
switch (selettore) {
    case '+':
    case '-':
    case '*':
    case '/':
        dx = pila.pop();
        sx = pila.pop();
        risultato = calcola(sx, dx, selettore);
        pila.push(risultato);
        break;
    default:
        //se non è un simbolo di operazione, dev'essere un numero
        Integer numero = new Integer(riga);
        pila.push(numero);
        break;
}

//lettura di un'altra riga
riga = in.readLine();
}

//preleva il risultato dalla pila e lo comunica
Integer risultato = pila.pop();
if (pila.empty())
    out.println("Il risultato è " + risultato);
else
    out.println("L'espressione contiene troppi operandi");
//fine blocco try
} catch (Exception e) {
    out.println("Errore: " + e.toString());
}
}

private static int calcola(int opSx, int opDx, char segno) {
    int ris = 0;
    switch (segno) {
        case '+':
            ris = opSx + opDx;
```

```
        break;
    case '-':
        ris = opSx - opDx;
        break;
    case '*':
        ris = opSx * opDx;
        break;
    case '/':
        ris = opSx / opDx;
        break;
    }
    return ris;
}

}
```

Ecco gli esempi precedenti, eseguiti con questa nuova versione dell'applicazione:

- (1) Inserire l'espressione da esaminare
(un elemento per riga, = per terminare)

```
5
4
+
0
/
Errore: java.lang.ArithmetricException: / by zero
```

- (2) Inserire l'espressione da esaminare
(un elemento per riga, = per terminare)

```
2
3
+
2ac
Errore: java.lang.NumberFormatException: For input string: "2ac"
```

- (3) Inserire l'espressione da esaminare
(un elemento per riga, = per terminare)

```
5
+
Errore: java.util.EmptyStackException
```

- (4) Inserire l'espressione da esaminare
(un elemento per riga, = per terminare)

```
6
```

```
3
4
+
=
L'espressione contiene troppi operandi
```

Nell'ultimo esempio l'esecuzione avviene tutta all'interno del blocco `try` perché non è sollevata alcuna eccezione: il messaggio d'errore viene fornito all'interno dell'ultima selezione nel blocco.

Negli altri tre casi, l'esecuzione del blocco `try` solleva invece un'eccezione che viene intercettata e trattata nel blocco `catch` successivo, all'interno del quale c'è l'istruzione che fornisce il messaggio d'errore. Si osservi che i messaggi forniti risultano inutili a un utente che non conosca il linguaggio Java e non sappia com'è stata implementata l'applicazione; essi forniscono informazioni legate all'anomalia verificatasi nel programma, ma non alla causa che l'ha provocata.

Scriviamo ora una nuova versione del metodo `main` in grado di trattare separatamente i tre tipi di eccezioni considerate.

A tale scopo è sufficiente introdurre differenti blocchi `catch` “specializzati” per ciascun tipo di eccezione. In particolare trattiamo le seguenti eccezioni.

- **ArithmeticException**

Viene sollevata quando si effettua una divisione intera per zero.

- **NumberFormatException**

Viene sollevata dal costruttore della classe `Integer`, con parametro di tipo `String`, se la stringa fornita non rappresenta un numero intero.

- **EmptyStackException**

Viene sollevata dal metodo `pop` della classe `Stack` quando si tenta di prelevare un elemento da una pila vuota.

La descrizione delle eccezioni che possono essere sollevate da un metodo o da un costruttore della libreria standard è disponibile nella documentazione della libreria stessa, insieme con la descrizione completa delle classi che rappresentano le eccezioni.

Ecco il codice della nuova versione della classe `Postfissa`. Per ogni tipo di eccezione forniamo un messaggio appropriato che indichi l'anomalia che provoca l'eccezione. Al fine di riprodurre il dato di input che ha generato il problema, nel caso di `NumberFormatException` invochiamo anche il metodo `toString` associato all'eccezione. Le eccezioni `NumberFormatException` e `ArithmeticException` sono definite nel package `java.lang`, e dunque non necessitano di importazione esplicita. Al contrario, la classe `EmptyStackException`, essendo definita in `java.util`, dev'essere importata esplicitamente.

```
import prog.io.*;
import java.util.Stack;
import java.util.EmptyStackException;
```

```
class Postfissa {

    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        //crea la pila
        Stack<Integer> pila = new Stack<Integer>();

        out.println("Inserire l'espressione da esaminare");
        out.println("(un elemento per riga, = per terminare)");
        String riga = in.readLine();
        try {
            //il codice che segue è esattamente quello della versione precedente
            while (!riga.equals("=")) {

                //elaborazione della riga letta
                Integer sx, dx, risultato;
                char selettore = riga.charAt(0);
                switch (selettore) {
                    case '+':
                    case '-':
                    case '*':
                    case '/':
                        dx = pila.pop();
                        sx = pila.pop();
                        risultato = calcola(sx, dx, selettore);
                        pila.push(risultato);
                        break;
                    default:
                        //se non è un simbolo di operazione, deve essere un numero
                        Integer numero = new Integer(riga);
                        pila.push(numero);
                        break;
                }
                //lettura di un'altra riga
                riga = in.readLine();
            }

            //preleva il risultato dalla pila e lo comunica
            Integer risultato = pila.pop();
        }
    }
}
```

```

        if (pila.empty())
            out.println("Il risultato è " + risultato);
        else
            out.println("L'espressione contiene troppi operandi");
        //fine blocco try
    } catch (ArithmaticException e) {
        out.println("Errore: divisione per zero");
    } catch (NumberFormatException e) {
        out.println("Input scorretto: " + e.toString());
    } catch (EmptyStackException e) {
        out.println("L'espressione contiene un numero insufficiente di operandi");
    }
}

private static int calcola(int opSx, int opDx, char segno) {
    int ris = 0;
    switch (segno) {
        case '+':
            ris = opSx + opDx;
            break;
        case '-':
            ris = opSx - opDx;
            break;
        case '*':
            ris = opSx * opDx;
            break;
        case '/':
            ris = opSx / opDx;
            break;
    }
    return ris;
}
}

```

Ecco di nuovo gli esempi precedenti, elaborati con questa nuova versione del metodo main:

- (1) Inserire l'espressione da esaminare
(un elemento per riga, = per terminare)

5

4

+

0

```

/
Errore: divisione per zero

(2) Inserire l'espressione da esaminare
    (un elemento per riga, = per terminare)
    2
    3
    +
    2ac
Input scorretto: java.lang.NumberFormatException: For input string: "2ac"

(3) Inserire l'espressione da esaminare
    (un elemento per riga, = per terminare)
    5
    +
L'espressione contiene un numero insufficiente di operandi

(4) Inserire l'espressione da esaminare
    (un elemento per riga, = per terminare)
    6
    3
    4
    +
    =
L'espressione contiene troppi operandi

```

Esercizi

- 11.6 Riscrivete l'ultima versione della classe Postfissa *senza utilizzare* il meccanismo delle eccezioni, ma introducendo, dove opportuno, costrutti di selezione. Si osservi che le modifiche da apportare sono piuttosto complesse e rendono il codice poco leggibile.
- 11.7 Riscrivete l'ultima versione della classe Postfissa in modo che riceva l'espressione su un'unica riga (servitevi di StringTokenizer).

11.5 Alcune eccezioni

In questo paragrafo riportiamo un elenco di alcune eccezioni definite nelle librerie standard di Java. Maggiori dettagli sono presentati nella documentazione delle librerie stesse. Come abbiamo detto, a ogni eccezione corrisponde una classe. Pertanto i nomi che elenchiamo sono nomi di classi dichiarate `public`. Tutte queste classi estendono, direttamente o indirettamente, la classe `Exception` definita nel package `java.lang`.

Le eccezioni indicate qui sono definite nel package `java.lang`.

- **ArithmeticException**

Viene sollevata qualora si verifichino condizioni eccezionali, come la divisione per zero, in operazioni aritmetiche.

- **ArrayIndexOutOfBoundsException**

Viene sollevata quando si tenta di accedere a una posizione inesistente di un array.

- **ClassCastException**

Viene sollevata quando si forza un tipo riferimento a un sottotipo di cui l'oggetto non è istanza. Ad esempio, eseguendo:

```
Rettangolo r;  
...  
Quadrato q = (Quadrato) r;
```

verrà sollevata l'eccezione nel caso in cui l'oggetto riferito da `r` non sia un'istanza di `Quadrato`.

- **NegativeArraySizeException**

Viene sollevata quando si tenta di creare un array di lunghezza negativa.

- **NullPointerException**

Viene sollevata quando si tenta di accedere a un oggetto tramite un riferimento `null`. Gli esempi più comuni sono l'invocazione di un metodo tramite un riferimento `null` o il tentativo di accedere a un elemento di un array nel caso il riferimento all'array sia `null`.

- **NumberFormatException**

Viene sollevata quando si tenta di convertire una stringa in un valore numerico senza che questa abbia il formato appropriato. Tale eccezione può essere sollevata, ad esempio, dal costruttore della classe `Integer` con argomento `String`.

- **StringIndexOutOfBoundsException**

Viene sollevata quando si tenta di accedere a una posizione inesistente in una stringa.

Le seguenti eccezioni sono definite nel package `java.util`.

- **EmptyStackException**

Viene sollevata quando si tenta di accedere a un elemento di uno `Stack` vuoto.

- **NoSuchElementException**

Viene sollevata quando si richiama il metodo `nextToken` di un'istanza di `StringTokenizer` e i token sono esauriti, o quando si richiama il metodo `next` di un oggetto che implementa l'interfaccia `Iterator` e non ci sono più elementi nell'iteratore.

Esercizi

11.8 Per ognuna delle eccezioni elencate nel paragrafo precedente scrivete un metodo `main` in cui essa possa essere sollevata. Riscrivete poi il metodo in maniera che l'eccezione venga intercettata.

11.9 Nel metodo `main` di ognuna delle seguenti classi può essere sollevata *almeno* un'eccezione. Per ogni metodo `main`:

- individuate le eccezioni che possono essere sollevate;
- riscrivete il metodo introducendo opportuni controlli in modo che le eccezioni individuate non vengano sollevate;
- riscrivete il metodo in modo che le eccezioni individuate vengano intercettate fornendo all'utente messaggi d'errore appropriati.

(1) `import prog.io.*;`

```
class Mcd {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int x = in.readInt("Primo numero? ");
        int y = in.readInt("Secondo numero? ");
        int resto;

        do {
            resto = x % y;
            x = y;
            y = resto;
        } while (resto != 0);

        out.println("Il massimo comun divisore è " + x);
    }
}
```

(2) `import prog.io.*;`

```
class Sequenza1 {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int quanti = in.readInt("Quanti numeri intendi inserire? ");
```

```
int[] numeri = new int[quanti];
for (int i = 0; i < quanti; i++)
    numeri[i] = in.readInt("Numero? ");

for (int i = 0; i < quanti; i++)
    out.println(numeri[i]);
}

}

(3) import prog.io.*;

class Sequenza2 {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        final int MAX = 10;
        int[] numeri = new int[MAX];
        int totNumeri = 0, n;

        while ((n = in.readInt("Numero? ")) != 0)
            numeri[totNumeri++] = n;

        out.println("I numeri inseriti sono:");
        for (int i = 0; i < totNumeri; i++)
            out.println(numeri[i]);

        int somma = 0;
        for (int i = 0; i < totNumeri; i++)
            somma += numeri[i];

        out.println("La loro somma è " + somma);
        out.println("La loro media è " + somma / totNumeri);
    }
}

(4) import prog.io.*;

class Sequenza3 {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        final int MAX = 10;
```

```

int[] numeri = new int[MAX];
int totNumeri = 0, n;

while ((n = in.readInt("Numero? ")) != 0)
    numeri[totNumeri++] = n;

out.println("I numeri inseriti sono:");
for (int i = 0; i < totNumeri; i++)
    out.println(numeri[i]);

int somma = 0;
for (int i = 0; i < totNumeri; i++)
    somma += numeri[i];

out.println("La loro somma è " + somma);
out.println("La loro media è " +
           (double) somma / totNumeri);
}
}

```

(5) import prog.io.*;

```

class Max1 {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        Integer numero, max = null;
        int n;

        while ((n = in.readInt("Numero? ")) != 0) {
            numero = new Integer(n);
            if (max == null || numero.compareTo(max) > 0)
                max = numero;
        }

        out.println("Il massimo è " + max.toString());
    }
}

```

(6) import prog.io.*;

```

class Max2 {

```

```

public static void main(String[] args) {
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();

    Integer numero, max = null, zero = new Integer(0);
    String s;

    while (!(numero = new Integer(in.readLine("Numero? "))).equals(zero)) {
        if (max == null || numero.compareTo(max) > 0)
            max = numero;
    }

    out.println("Il massimo è " + max.toString());
}

(7) import prog.io.*;

class MaxPari {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        Integer numero, max = null;
        int n;

        while ((n = in.readInt("Numero? ")) != 0) {
            numero = new Integer(n);
            if (n % 2 == 0 && (max == null ||
                numero.compareTo(max) > 0))
                max = numero;
        }

        out.println("Il massimo numero pari è " + max.toString());
    }
}

(8) import prog.io.*;

class MaxMultiplo {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

```

```

        Integer numero, max = null;
        int n;
        int d = in.readInt("Inserire il numero dei cui multipli " +
                            "si vuole il massimo ");

        while ((n = in.readInt("Numero? ")) != 0) {
            numero = new Integer(n);
            if (n % d == 0 && (max == null ||
                                   numero.compareTo(max) > 0))
                max = numero;
        }

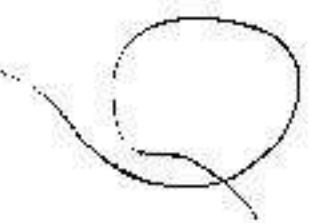
        out.println("Il massimo multiplo di " + d + " è " +
                    max.toString());
    }
}

```

11.6 Come sollevare le eccezioni: l'istruzione throw

In questo paragrafo mostreremo come sia possibile sollevare esplicitamente un'eccezione nel codice di un metodo o di un costruttore.

Molti esempi di questo paragrafo e dei successivi sono basati sulla classe `Frazione`. Considereremo in particolare l'ultima versione della classe che abbiamo sviluppato, in cui il costruttore con due parametri memorizza la frazione in forma semplificata. Se come denominatore viene fornito zero, tale costruttore memorizza zero sia nel campo `num` sia nel campo `den`, con la convenzione che ciò rappresenta un valore impossibile. Riportiamo per comodità il testo del costruttore:



```

public Frazione(int x, int y) {
    if (y == 0) {
        num = 0;
        den = 0;
    } else {
        //memorizza il segno
        boolean negativo = (x < 0 && y > 0) || (x > 0 && y < 0);

        if (x < 0)
            x = - x; //elimina l'eventuale segno meno davanti a x
        if (y < 0)
            y = - y; //elimina l'eventuale segno meno davanti a y
    }
}

```

```

int m = mcd(x, y);
if (negativo)
    num = - x / m; //il segno meno viene memorizzato al numeratore
else
    num = x / m;
den = y / m;
}
}

```

Ricordiamo che qualora l'oggetto non rappresenti una frazione, il metodo `toString` fornisce come risultato la stringa "impossibile".

Consideriamo la seguente classe di prova, che legge i dati relativi a due frazioni e ne calcola il quoziente.⁶

```

import prog.io.*;

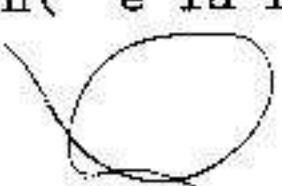
class Prova {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int n = in.readInt("Numeratore prima frazione? ");
        int d = in.readInt("Denominatore prima frazione? ");
        Frazione f1 = new Frazione(n, d);
        out.println("E' stata inserita la frazione " + f1.toString());

        n = in.readInt("Numeratore seconda frazione? ");
        d = in.readInt("Denominatore seconda frazione? ");
        Frazione f2 = new Frazione(n, d);
        out.println("E' stata inserita la frazione " + f2.toString());

        Frazione f3 = f1.diviso(f2);
        out.print("Il quoziente di " + f1.toString() + " e " +
                  f2.toString());
        out.println(" è la frazione " + f3.toString());
    }
}

```



Seguono tre esempi di esecuzione dell'applicazione: nel primo non ci sono problemi; nel secondo la seconda frazione vale zero, e dunque la divisione produce un risultato impossibile; nel terzo la prima frazione viene inserita con denominatore uguale a zero.

⁶ Per semplicità, in tutto il capitolo supporremo che la classe `Frazione` e le altre classi introdotte siano poste nella stessa directory della classe `Prova`, senza collocarle in un package.

```
Numeratore prima frazione? 3
Denominatore prima frazione? 4
E' stata inserita la frazione 3/4
Numeratore seconda frazione? 2
Denominatore seconda frazione? 5
E' stata inserita la frazione 2/5
Il quoziente di 3/4 e 2/5 è la frazione 15/8
```

```
Numeratore prima frazione? 3
Denominatore prima frazione? 4
E' stata inserita la frazione 3/4
Numeratore seconda frazione? 0
Denominatore seconda frazione? 5
E' stata inserita la frazione 0
Il quoziente di 3/4 e 0 è la frazione impossibile
```

```
Numeratore prima frazione? 3
Denominatore prima frazione? 0
E' stata inserita la frazione impossibile
Numeratore seconda frazione? 2
Denominatore seconda frazione? 5
E' stata inserita la frazione 2/5
Il quoziente di impossibile e 2/5 è la frazione impossibile
```

Vogliamo modificare il costruttore della classe `Frazione` in modo che quando il secondo argomento, cioè quello corrispondente al denominatore, è zero sollevi un'eccezione.

Ricordiamo che le eccezioni sono oggetti: dobbiamo decidere, prima di tutto, a quale classe vogliamo che appartenga l'eccezione che intendiamo sollevare. Possiamo definire a questo scopo una nuova classe, o utilizzarne una già esistente. Cominciamo dalla seconda alternativa: poiché l'evento anomalo che stiamo descrivendo si riferisce a oggetti legati all'aritmetica, decidiamo di descriverlo utilizzando la classe `ArithmeticException`.

Per sollevare l'eccezione è necessario costruire l'oggetto che la descrive. Dunque dovremo invocare un opportuno costruttore della classe `ArithmeticException`. In particolare tale classe dispone di un costruttore che riceve come argomento una stringa corrispondente a un messaggio d'errore da associare all'evento anomalo. Il costruttore viene invocato, come sempre, in un'espressione `new`:

```
new ArithmeticException("Frazione non valida: denominatore 0")
```

Per sollevare l'eccezione si utilizza l'istruzione `throw` (solleva) seguita dal riferimento all'oggetto eccezione. In sostanza possiamo scrivere:

```
throw new ArithmeticException("Frazione non valida: denominatore 0")
```

Ecco il codice del costruttore di `Frazione` scritto in maniera che sollevi l'eccezione qualora il secondo argomento sia uguale a zero.

```

public Frazione(int x, int y) {
    if (y == 0)
        throw new ArithmeticException("Frazione non valida: " +
                                       "denominatore 0");
    else {
        //memorizza il segno
        boolean negativo = (x < 0 && y > 0) || (x > 0 && y < 0);

        if (x < 0)
            x = - x; //elimina l'eventuale segno meno davanti a x
        if (y < 0)
            y = - y; //elimina l'eventuale segno meno davanti a y

        int m = mcd(x, y);
        if (negativo)
            num = - x / m; //il segno meno viene memorizzato al numeratore
        else
            num = x / m;
        den = y / m;
    }
}

```

Ecco quanto si ottiene per gli ultimi due esempi precedenti eseguendo la classe Prova con questa nuova versione di **Frazione**.

```

Numeratore prima frazione? 3
Denominatore prima frazione? 4
E' stata inserita la frazione 3/4
Numeratore seconda frazione? 0
Denominatore seconda frazione? 5
E' stata inserita la frazione 0
Exception in thread "main" java.lang.ArithmetricException:
Frazione non valida: denominatore 0
    at Frazione.<init>(Frazione.java:7)
    at Frazione.diviso(Frazione.java:51)
    at Prova.main(Prova.java:18)

```

```

Numeratore prima frazione? 3
Denominatore prima frazione? 0
Exception in thread "main" java.lang.ArithmetricException:
Frazione non valida: denominatore 0
    at Frazione.<init>(Frazione.java:7)
    at Prova.main(Prova.java:10)

```

Si osservino i messaggi di errore: essi riportano il nome dell'eccezione seguito dalla stringa che abbiamo fornito come parametro al costruttore dell'oggetto eccezione dopo l'istruzione `throw` nel costruttore di `Frazione`.

Come evidenziato dal messaggio d'errore, nel primo esempio l'eccezione è stata sollevata dal costruttore di `Frazione` (indicato nel messaggio d'errore con `Frazione.<init>`), richiamato dal metodo `diviso`, richiamato a sua volta dal metodo `main` di `Prova`. In particolare il costruttore ha rinviato l'eccezione a `diviso`, che a sua volta l'ha rinviata a `main` che, infine, l'ha rinviata alla Java Virtual Machine.

Nel secondo esempio l'eccezione è stata sollevata dal costruttore, richiamato direttamente da `main`.

Presentiamo ora una nuova versione della classe di prova in cui intercettiamo l'eccezione mediante l'istruzione `try-catch` e visualizziamo la stringa prodotta dal metodo `toString` dell'oggetto eccezione:

```
import prog.io.*;

class Prova {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int n = in.readInt("Numeratore prima frazione? ");
        int d = in.readInt("Denominatore prima frazione? ");
        try {
            Frazione f1 = new Frazione(n, d);
            out.println("E' stata inserita la frazione " + f1.toString());

            n = in.readInt("Numeratore seconda frazione? ");
            d = in.readInt("Denominatore seconda frazione? ");
            Frazione f2 = new Frazione(n, d);
            out.println("E' stata inserita la frazione " + f2.toString());

            Frazione f3 = f1.diviso(f2);
            out.print("Il quoziente di " + f1.toString() + " e " +
                     f2.toString());
            out.println(" è la frazione " + f3.toString());
        } catch (ArithmeticException e) {
            out.println(e.toString());
        }
    }
}
```

Ecco ciò che si ottiene dalle esecuzioni con i dati dei due esempi precedenti:

```
Numeratore prima frazione? 3
Denominatore prima frazione? 4
E' stata inserita la frazione 3/4
Numeratore seconda frazione? 0
Denominatore seconda frazione? 5
E' stata inserita la frazione 0
java.lang.ArithmetricException: Frazione non valida: denominatore 0
```

```
Numeratore prima frazione? 3
Denominatore prima frazione? 0
java.lang.ArithmetricException: Frazione non valida: denominatore 0
```

Si noti il risultato del metodo `toString`: contiene il nome dell'eccezione seguito dal messaggio che abbiamo fornito al costruttore.

Intercettando l'eccezione avremmo potuto utilizzare altri metodi. Ad esempio, volendo visualizzare solo il messaggio senza il nome dell'eccezione, anziché invocare `toString` avremmo potuto invocare il metodo `getMessage` ereditato dalla classe `Throwable`. Volendo evidenziare il punto in cui è stata sollevata l'eccezione, avremmo potuto ricorrere al metodo `printStackTrace`, anch'esso ereditato da `Throwable`. Ad esempio, se la parte `catch` fosse scritta come:

```
... catch (ArithmetricException e) {
    out.println(e.getMessage());
    e.printStackTrace();
}
```

una possibile esecuzione sarebbe:

```
Numeratore prima frazione? 6
Denominatore prima frazione? 0
Frazione non valida: denominatore 0
java.lang.ArithmetricException: Frazione non valida: denominatore 0
at Frazione.<init>(Frazione.java:6)
at Prova.main(Prova.java:11)
```

11.7 Come definire un'eccezione

Nell'esempio precedente abbiamo utilizzato `ArithmetricException` per rappresentare il tentativo di costruire una frazione con denominatore uguale a zero.

Vogliamo ora definire un'apposita classe che rappresenti questa particolare anomalia legata alle frazioni. Ricordiamo che gli oggetti che descrivono le eccezioni sono istanze della sottoclasse `Exception` di `Throwable`. Dunque per definire una nuova eccezione dobbiamo estendere

`Exception` o una delle sue sottoclassi. La scelta di quale classe estendere dipende da ciò che si vuole rappresentare con l'eccezione e dal fatto che si voglia definire un'*eccezione controllata* oppure un'*eccezione non controllata*. Discuteremo la differenza tra eccezioni controllate e non controllate più avanti nel capitolo. Per ora è sufficiente osservare che l'eccezione che vogliamo definire è legata all'aritmetica delle frazioni; quindi può essere pensata come una particolare eccezione di tipo aritmetico. Per questa ragione decidiamo di estendere `ArithmeticException` definendo:

```
public class FrazioneException extends ArithmeticException
```

Quale dev'essere il comportamento degli elementi della classe? Per ora ci limitiamo a definire un comportamento estremamente semplice, identico a quello mostrato prima per `ArithmeticException`: dev'essere possibile associare all'oggetto eccezione un messaggio che sia poi visualizzato dal metodo `toString`.⁷ Dunque la classe dovrà disporre di un costruttore che riceva, tramite il parametro, il messaggio associato all'errore:

```
public FrazioneException(String msg)
```

Dove memorizzeremo il messaggio? Abbiamo appena ricordato che la superclasse `ArithmeticException` è in grado di fornire lo stesso comportamento. Non è quindi necessario introdurre nuovi campi, possiamo basarci esclusivamente sul meccanismo dell'ereditarietà: la costruzione di un'istanza di `FrazioneException`, con un particolare messaggio, verrà delegata al costruttore di `ArithmeticException` cui sarà fornito il medesimo messaggio:

```
public FrazioneException(String msg) {
    super(msg);
}
```

La classe eredita, inoltre, i metodi della superclasse, tra cui `toString`. Pertanto non è necessario aggiungere nulla. Il testo completo della classe è dunque:

```
public class FrazioneException extends ArithmeticException {
    public FrazioneException(String msg) {
        super(msg);
    }
}
```

Nel costruttore di `Frazione`, per sollevare `FrazioneException` dobbiamo:

- costruire un'istanza dell'eccezione richiamando il costruttore in un'espressione `new`
- scrivere in un'istruzione `throw` il riferimento all'oggetto ottenuto.

⁷ La maggior parte delle classi che rappresentano eccezioni si limitano a comportamenti di questo tipo, oltre, naturalmente, a quelli creditati dalle superclassi.

Ecco la nuova versione del costruttore:

```
public Frazione(int x, int y) {
    if (y == 0)
        throw new FrazioneException("Frazione non valida: " +
                                      "denominatore 0");
    else {
        //memorizza il segno
        boolean negativo = (x < 0 && y > 0) || (x > 0 && y < 0);

        if (x < 0)
            x = - x; //elimina l'eventuale segno meno davanti a x
        if (y < 0)
            y = - y; //elimina l'eventuale segno meno davanti a y

        int m = mcd(x, y);
        if (negativo)
            num = - x / m; //il segno meno viene memorizzato al numeratore
        else
            num = x / m;
        den = y / m;
    }
}
```

In base alla gerarchia delle classi, un'istanza di `FrazioneException` è anche istanza di `ArithmetiException`. Di conseguenza l'ultima versione della classe `Prova` presentata è in grado di intercettare, senza alcuna modifica, eccezioni di tipo `FrazioneException`. Tuttavia è consigliabile intercettare le eccezioni indicando il loro tipo specifico:⁸

```
import prog.io.*;
class Prova {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int n = in.readInt("Numeratore prima frazione? ");
        int d = in.readInt("Denominatore prima frazione? ");
        try {
            Frazione f1 = new Frazione(n, d);
            out.println("E' stata inserita la frazione " + f1.toString());
        }
    }
}
```

⁸ Ricordiamo che ciò diventa necessario quando possono essere sollevate eccezioni di tipo diverso, da trattare in maniera differenziata. Si veda a questo riguardo l'esempio sul calcolo di espressioni in notazione postfissa, presentato in questo capitolo.

```
n = in.readInt("Numeratore seconda frazione? ");
d = in.readInt("Denominatore seconda frazione? ");
Frazione f2 = new Frazione(n, d);
out.println("E' stata inserita la frazione " + f2.toString());

Frazione f3 = f1.diviso(f2);
out.print("Il quoziente di " + f1.toString() + " e "
        + f2.toString());
out.println(" è la frazione " + f3.toString());
} catch (FrazioneException e) {
    out.println(e.toString());
}
}
```

```
Numeratore prima frazione? 3
Denominatore prima frazione? 0
FrazioneException: Frazione non valida: denominatore 0
```

Nelle classi che rappresentano eccezioni è possibile definire liberamente campi e metodi come in tutte le altre classi.

A titolo d'esempio presentiamo una nuova versione della classe `FrazioneException` in cui, oltre a un messaggio associato all'errore, viene memorizzato un valore intero uguale al numeratore della frazione che si desiderava costruire.

A tale scopo definiamo un campo di tipo int. La classe ha due costruttori: uno che riceve il messaggio da memorizzare e l'intero, e l'altro che riceve solo l'intero. Forniamo inoltre un metodo `getNumeratore` per conoscere il numeratore della frazione “mancata”:

```
public class FrazioneException extends ArithmeticException {  
    private int num;  
  
    public FrazioneException(String msg, int num) {  
        super(msg);  
        this.num = num;  
    }  
  
    public FrazioneException(int num) {  
        this.num = num;  
    }  
}
```

```

public int getNumeratore() {
    return num;
}
}

```

Si osservi che, in base alle regole sui costruttori, all'inizio del costruttore con un argomento c'è una chiamata al costruttore privo di argomenti della superclasse; il codice è compilato correttamente perché `ArithmetricException` possiede anche un costruttore privo di argomenti.

Riscriviamo ora il costruttore di `Frazione` in modo che utilizzi la nuova versione di `FrazioneException`:

```

public Frazione(int x, int y) {
    if (y == 0)
        throw new FrazioneException("Frazione non valida: " +
                                      "denominatore 0", x);
    else {
        //memorizza il segno
        boolean negativo = (x < 0 && y > 0) || (x > 0 && y < 0);

        if (x < 0)
            x = - x; //elimina l'eventuale segno meno davanti a x
        if (y < 0)
            y = - y; //elimina l'eventuale segno meno davanti a y

        int m = mcd(x, y);
        if (negativo)
            num = - x / m; //il segno meno viene memorizzato al numeratore
        else
            num = x / m;
        den = y / m;
    }
}

```

Ecco una nuova versione della classe `Prova` in cui viene utilizzato il metodo `getNumeratore` dell'eccezione:

```

import prog.io.*;

class Prova {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

```

```

int n = in.readInt("Numeratore prima frazione? ");
int d = in.readInt("Denominatore prima frazione? ");
try {
    Frazione f1 = new Frazione(n, d);
    out.println("E' stata inserita la frazione " + f1.toString());

    n = in.readInt("Numeratore seconda frazione? ");
    d = in.readInt("Denominatore seconda frazione? ");
    Frazione f2 = new Frazione(n, d);
    out.println("E' stata inserita la frazione " + f2.toString());

    Frazione f3 = f1.diviso(f2);
    out.print("Il quoziente di " + f1.toString() + " e " +
              f2.toString());
    out.println(" è la frazione " + f3.toString());
} catch (FrazioneException e) {
    out.println(e.toString());
    out.println("Il valore del numeratore è " + e.getNumeratore());
}
}
}

```

Segue un esempio di esecuzione:

```

Numeratore prima frazione? 3
Denominatore prima frazione? 4
E' stata inserita la frazione 3/4
Numeratore seconda frazione? 0
Denominatore seconda frazione? 5
E' stata inserita la frazione 0
FrazioneException: Frazione non valida: denominatore 0
Il valore del numeratore è 3

```

Presentiamo infine un esempio più articolato in cui utilizziamo due costrutti try-catch, in questo caso innestati, per trattare separatamente i due possibili malfunzionamenti:

```

import prog.io.*;

class Prova {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int n = in.readInt("Numeratore prima frazione? ");

```

```

int d = in.readInt("Denominatore prima frazione? ");
try {
    Frazione f1 = new Frazione(n, d);
    out.println("E' stata inserita la frazione " + f1.toString());

    n = in.readInt("Numeratore seconda frazione? ");
    d = in.readInt("Denominatore seconda frazione? ");
    Frazione f2 = new Frazione(n, d);
    out.println("E' stata inserita la frazione " + f2.toString());

    try {
        Frazione f3 = f1.diviso(f2);
        out.print("Il quoziente di " + f1.toString() + " e " +
                 f2.toString());
        out.println(" è la frazione " + f3.toString());
    } catch (FrazioneException e) {
        out.println("La divisione di " + f1.toString() + " e " +
                   f2.toString() + " è impossibile:");
        out.print(" per il numeratore si è ottenuto " +
                 e.getNumeratore());
        out.println(", ma per il denominatore si è ottenuto 0");
    }
} catch (FrazioneException e) {
    out.println("Errore: è stata inserita una frazione con " +
               "denominatore zero");
}
}
}

```

Seguono due esempi di esecuzione:

```

Numeratore prima frazione? 3
Denominatore prima frazione? 0
Errore: è stata inserita una frazione con denominatore zero

```

```

Numeratore prima frazione? 3
Denominatore prima frazione? 4
E' stata inserita la frazione 3/4
Numeratore seconda frazione? 0
Denominatore seconda frazione? 5
E' stata inserita la frazione 0
La divisione di 3/4 e 0 è impossibile:
    per il numeratore si è ottenuto 3, ma per il denominatore si
    è ottenuto 0

```

Esercizi

- 11.10 Considerate la classe `Orario` implementata nel Capitolo 7. Qualora la stringa non abbia il formato corretto, il costruttore con argomento `String` di tale classe solleva eccezioni. Queste eccezioni non sono sollevate esplicitamente dal costruttore, ma dai metodi `parseInt` e `substring`. Definite una nuova eccezione `OrarioException` per rappresentare anomalie relative agli orari. Riscrivete poi il costruttore di `Orario` con parametro `String` in modo che, in caso di stringa in formato scorretto, sollevi una `OrarioException` (per risolvere il problema è utile che il costruttore intercetti le eccezioni sollevate dai metodi chiamati sollevando a sua volta `OrarioException`).
- 11.11 Riscrivete i costruttori della classe `Orario` in modo che sollevino una `OrarioException` tutte le volte che si tenta di costruire un oggetto con uno stato inconsistente (ad esempio, se il numero dei minuti è negativo o maggiore di 59).
- 11.12 Considerate la seguente classe:

```
public class EccezioneArray extends RuntimeException {
    public EccezioneArray(String s) {
        super(s);
    }
}
```

Codificate un metodo (che sarà collocato in un'altra classe), con il prototipo

```
public static int nDispari(int[] h)
```

il cui compito è quello di contare e restituire il numero di valori dispari presenti nell'array di cui viene fornito il riferimento tramite il parametro. Qualora il riferimento sia null, il metodo deve sollevare una `EccezioneArray`.

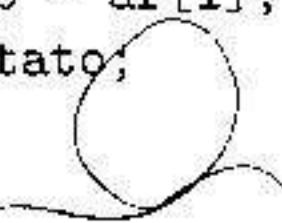
Si consideri poi il seguente metodo `m`, posto nella stessa classe di `nDispari`. Il metodo `m` richiama un metodo `leggi` e il metodo `nDispari`, e infine visualizza il valore restituito da quest'ultimo:

```
public static void m() {
    int[] x = leggi();
    int y = nDispari(x);
    System.out.println(y);
}
```

Riscrivete il codice di `m` in modo che, nel caso `nDispari` sollevi l'eccezione `EccezioneArray`, in luogo del valore restituito da `nDispari`, venga visualizzato un messaggio d'errore.

- 11.13 Considerate ancora la classe `EccezioneArray` e il seguente metodo statico che riceve, come parametro, un array di tipo `Rettangolo` e restituisce il riferimento all'oggetto di altezza maggiore presente nell'array:

```
public static Rettangolo rettangoloAltezzaMax(Rettangolo[] ar) {
    Rettangolo risultato = ar[0];
    for (int i = 1; i < ar.length; i++)
        if (ar[i].getAltezza() > risultato.getAltezza())
            risultato = ar[i];
    return risultato;
}
```



Durante l'esecuzione del metodo possono verificarsi tre circostanze eccezionali (argomento `null`, array vuoto e presenza di riferimenti `null` nell'array). Riscrivete il metodo *senza* utilizzare costrutti `try-catch`, ma introducendo esclusivamente istruzioni di selezione, in modo che in queste situazioni sollevi eccezioni di tipo `EccezioneArray` con opportuni messaggi d'errore.

Si consideri poi il seguente metodo, posto nella stessa classe del metodo `rettangoloAltezzaMax`:

```
public static void visualizzaRettangoloAltezzaMax(Rettangolo[] p) {
    Rettangolo v = rettangoloAltezzaMax(p);
    System.out.println(v.toString());
}
```

Riscrivete il codice di `visualizzaRettangoloAltezzaMax` in modo che, nel caso `rettangoloAltezzaMax` sollevi l'eccezione `EccezioneArray`, in luogo della stringa restituita da `v.toString()` venga visualizzato il messaggio d'errore associato all'eccezione.

- 11.14 Indicate il risultato restituito da ognuno dei seguenti metodi qualora venga fornito come argomento il valore 3:

- ```
public static int f(int n) {
 if (n % 2 == 0 || n < 3)
 return n;
 else
 return 3 * f(n / 2) + 2 * f(n + 3);
}
```

```
• public static int g(int n) {
 try {
 return 4 * g((n + 1) / (n - 2));
 } catch (Exception e) {
 return n;
 }
}
```

11.15 Indicate il risultato restituito da ognuno dei seguenti metodi qualora venga fornito come argomento il valore 6:

```
• public static int f(int n) {
 if (n % 2 == 1 || n < 3)
 return n;
 else
 return f(n - 2) + f(n + 1);
}

• public static int g(int n) {
 try {
 return 3 * g(n / (n - 1));
 } catch (Exception e) {
 return n;
 }
}

• public static int f(int n) {
 if (n <= 3)
 return n;
 else
 return f(n - 1) + f(n - 2);
}

• public static int g(int n) {
 try {
 return 2 * g((n + 1) / ((n + 1) % 2));
 } catch (Exception e) {
 return n;
 }
}
```

11.16 Indicate il risultato restituito da ognuno dei seguenti metodi qualora venga fornito come argomento il valore 5:

```
• public static int f(int n) {
 if (n <= 2)
```

```

 return n;
}
else
 return f(n - 1) * f(n - 2);
}

• public static int g(int n) {
 try {
 return g((n - 1) / (n % 2)) + 2;
 } catch (Exception e) {
 return n;
 }
}

```

## 11.8 Eccezioni controllate e non controllate

Abbiamo visto che le eccezioni vengono definite estendendo la sottoclasse `Exception` di `Throwable`. Le eccezioni del linguaggio Java sono suddivise in due gruppi fondamentali.

- *Eccezioni non controllate*

Sono tutte le istanze di `RuntimeException`, cioè tutte le eccezioni definite dalla classe `RuntimeException` e dalle sue sottoclassi (dirette o indirette).

- *Eccezioni controllate*

Sono tutte le altre, cioè tutte le istanze di `Exception` che non sono istanze di `RuntimeException`. In altre parole sono tutte le eccezioni definite direttamente dalla classe `Exception` e da tutte le sue sottoclassi (dirette o indirette) che non sono sottoclassi anche di `RuntimeException`.

Si osservi che tutte le eccezioni utilizzate negli esempi presentati fin qui sono non controllate.

Spieghiamo ora il significato della terminologia utilizzata. I due gruppi di eccezioni si differenziano per il modo in cui vengono trattate dal compilatore. In particolare il compilatore *controlla* che ogni metodo o costruttore che possa sollevare un'eccezione controllata ne fornisca un trattamento esplicito; in caso contrario il compilatore segnala un errore. Il trattamento dell'eccezione può avvenire in uno dei seguenti modi:

- *intercettandola* tramite l'istruzione `try-catch`;
- *delegandola esplicitamente* al chiamante mediante un'opportuna dichiarazione introdotta dalla parola chiave `throws` nell'intestazione del metodo (o costruttore) stesso, come vedremo più avanti.

Per le eccezioni non controllate, invece, il compilatore non effettua controlli: come mostrato negli esempi precedenti, quando non sono intercettate, esse vengono automaticamente delegate al chiamante senza bisogno di alcuna indicazione esplicita nel codice.

Per illustrare le differenze nell'uso delle eccezioni controllate rispetto a quelle non controllate, modifichiamo l'esempio precedente definendo `FrazioneException` come controllata. A tale scopo estendiamo direttamente `Exception`:

```
public class FrazioneException extends Exception {
 private int num;

 public FrazioneException(String msg, int num) {
 super(msg);
 this.num = num;
 }

 public FrazioneException(int num) {
 this.num = num;
 }

 public int getNumeratore() {
 return num;
 }
}
```

La classe `FrazioneException` viene compilata correttamente. Al contrario, ricompilando la classe `Frazione` otteniamo il seguente messaggio d'errore:

```
Frazione.java:7: unreported exception FrazioneException;
must be caught or declared to be thrown
 throw new FrazioneException("Frazione non valida: denominatore 0", x);
^
1 error
```

Il messaggio segnala un errore alla riga numero 7. Essa corrisponde all'istruzione del costruttore con due argomenti nella quale, nel caso di secondo argomento zero, viene sollevata l'eccezione. Il messaggio segnala in particolare che l'eccezione dev'essere intercettata o dichiarata esplicitamente come sollevabile. Questo tipo di messaggio, che non veniva fornito con la versione precedente di `FrazioneException`, è legato appunto al "controllo" effettuato dal compilatore sulle eccezioni controllate.

Poiché non siamo interessati a intercettare l'eccezione nel costruttore, ma vogliamo proprio che esso la deleghi al codice che l'ha richiamato, dichiariamo esplicitamente che essa può essere sollevata. Questa dichiarazione riguarda il comportamento del costruttore rispetto a chi lo chiama; pertanto viene indicata nell'intestazione del costruttore scrivendo, dopo la lista dei parametri, la parola riservata `throws` seguita dal nome dell'eccezione:

```
public Frazione(int x, int y) throws FrazioneException
```

A questo punto il comportamento del costruttore rispetto al codice chiamante è definito in maniera più precisa: la chiamata del costruttore può produrre l'oggetto o sollevare un'eccezione del tipo indicato. Il resto del codice del costruttore non necessita di alcuna modifica.

Riportiamo, per comodità, il testo completo della classe `Frazione` dopo questa modifica:

```
public class Frazione implements Comparable<Frazione> {
 private int num; // il numeratore della frazione
 private int den; // il denominatore della frazione

 public Frazione(int x, int y) throws FrazioneException {
 if (y == 0)
 throw new FrazioneException("Frazione non valida: " +
 "denominatore 0", x);

 else {
 //memorizza il segno
 boolean negativo = (x < 0 && y > 0) || (x > 0 && y < 0);

 if (x < 0)
 x = - x; //elimina l'eventuale segno meno davanti a x
 if (y < 0)
 y = - y; //elimina l'eventuale segno meno davanti a y

 int m = mcd(x, y);
 if (negativo)
 num = - x / m; //il segno meno viene memorizzato al numeratore
 else
 num = x / m;
 den = y / m;
 }
 }

 public Frazione(int x) {
 this(x, 1);
 }

 public Frazione piu(Frazione f) {
 int n = this.num * f.den + this.den * f.num;
 int d = this.den * f.den;
 return new Frazione(n, d);
 }

 public Frazione meno(Frazione f) {
 int n = this.num * f.den - this.den * f.num;
```

```
int d = this.den * f.den;
return new Frazione(n, d);
}

public Frazione per(Frazione f) {
 int n = this.num * f.num;
 int d = this.den * f.den;
 return new Frazione(n, d);
}

public Frazione diviso(Frazione f) {
 int n = this.num * f.den;
 int d = this.den * f.num;
 return new Frazione(n, d);
}

public boolean equals(Frazione f) {
 return this.num == f.num && this.den == f.den;
}

public boolean equals(Object o) {
 if (o instanceof Frazione)
 return this.equals((Frazione) o);
 else
 return false;
}

public boolean isMinore(Frazione f) {
 Frazione g = this.meno(f);
 return g.num < 0;
}

public boolean isMaggiore(Frazione f) {
 Frazione g = this.meno(f);
 return g.num > 0;
}

public String toString() {
 if (den == 0)
 return "impossibile";
 else if (den == 1)
 return String.valueOf(num);
```

```
 else
 return num + "/" + den;
 }

 public int getNumeratore() {
 return num;
 }

 public int getDenominatore() {
 return den;
 }

 public int compareTo(Frazione altra) {
 if (this.equals(altra))
 return 0;
 else if (this.isMinore(altra))
 return -1;
 else
 return 1;
 }

 private static int mcd(int a, int b) {
 int resto;
 do {
 resto = a % b;
 a = b;
 b = resto;
 } while (resto != 0);
 return a;
 }
}
```

Compilando questa versione di Frazione otteniamo nuovamente segnalazioni di errore:

Frazione.java:27: unreported exception FrazioneException;

must be caught or declared to be thrown

    this(x, 1);

    ^

Frazione.java:33: unreported exception FrazioneException;

must be caught or declared to be thrown

    return new Frazione(n, d);

    ^

Frazione.java:39: unreported exception FrazioneException;

must be caught or declared to be thrown

```

 return new Frazione(n, d);

Frazione.java:45: unreported exception FrazioneException;
must be caught or declared to be thrown
 return new Frazione(n, d);

Frazione.java:51: unreported exception FrazioneException;
must be caught or declared to be thrown
 return new Frazione(n, d);

5 errors

```

I cinque errori segnalati si riferiscono alle chiamate del costruttore con due argomenti, in quanto ognuna di esse può sollevare una `FrazioneException` (come indicato in maniera esplicita dalla clausola `throws` del costruttore stesso). Dunque i metodi o costruttori che richiamano il costruttore con due argomenti devono, a loro volta, trattare l'eccezione intercettandola o rinviandola al chiamante.

In particolare i cinque errori indicati corrispondono al costruttore con un argomento, che richiama quello con due utilizzando `this`, e ai metodi `piu`, `meno`, `per` e `diviso`, che richiamano il costruttore con due argomenti per costruire la frazione corrispondente ai risultati dei calcoli. In ognuno dei cinque casi dobbiamo di nuovo decidere se intercettare l'eccezione o delegarla al chiamante.

Iniziamo dal metodo `diviso`. In questo caso la scelta più ragionevole è quella di delegare l'eccezione: se il risultato della divisione non è una frazione, il metodo `diviso` solleva un'eccezione o, più precisamente, rinvia a chi l'ha chiamato l'eccezione sollevata dal costruttore. Pertanto è sufficiente dichiarare l'eccezione nell'intestazione del metodo:<sup>9</sup>

```

public Frazione diviso(Frazione f) throws FrazioneException {
 int n = this.num * f.den;
 int d = this.den * f.num;
 return new Frazione(n, d);
}

```

Nel caso degli altri metodi di calcolo, è facile osservare che il risultato dell'operazione effettuata sarà sempre una frazione. Quindi, di fatto, la chiamata del costruttore in questi metodi non potrà mai sollevare l'eccezione. Tuttavia il compilatore non conosce la logica di ciò che stiamo scrivendo: si limita a osservare che questi metodi richiamano il costruttore con due argomenti, nella cui intestazione è indicata la possibilità di sollevare `FrazioneException`. Pertanto il compilatore impone di trattare l'eccezione nei metodi.

In base a quanto abbiamo appena osservato, non ha molto senso che questi metodi deleghi no un'eccezione, che non potrà mai essere sollevata concretamente durante la loro esecuzione.

---

<sup>9</sup> Quando nell'intestazione di un metodo o di un costruttore è necessario dichiarare più di un'eccezione, è sufficiente utilizzare un'unica istruzione `throws` seguita dai nomi delle eccezioni separati da una virgola.

Dunque decidiamo di trattare l'eccezione nei metodi introducendo un costrutto `try-catch`, dove nella parte `try` costruiamo e restituiamo la nuova frazione, mentre nella parte `catch` (che, lo ripetiamo, di fatto non sarà mai eseguita) restituiamo `null` (questo per garantire che i metodi restituiscano in ogni caso un riferimento di tipo `Frazione`, secondo quanto indicato nel prototipo). Riportiamo come esempio il metodo `per`, riscritto secondo questo schema:

```
public Frazione per(Frazione f) {
 int n = this.num * f.num;
 int d = this.den * f.den;
 try {
 return new Frazione(n, d);
 } catch (FrazioneException e) {
 return null;
 }
}
```

Consideriamo infine il costruttore con un solo argomento. Esso deve costruire una frazione con denominatore uguale a 1. Anche in questo caso, dunque, la chiamata del costruttore con due argomenti non potrà mai sollevare l'eccezione. La soluzione più semplice consiste nel riscrivere il costruttore con un argomento direttamente, senza invocare quello con due:

```
public Frazione(int x) {
 num = x;
 den = 1;
}
```

Ecco il codice completo della classe `Frazione` dopo queste modifiche, compilabile senza errori:

```
public class Frazione implements Comparable<Frazione> {
 private int num; // il numeratore della frazione
 private int den; // il denominatore della frazione

 public Frazione(int x, int y) throws FrazioneException {
 if (y == 0)
 throw new FrazioneException("Frazione non valida: " +
 "denominatore 0", x);

 else {
 //memorizza il segno
 boolean negativo = (x < 0 && y > 0) || (x > 0 && y < 0);

 if (x < 0)
 x = - x; //elimina l'eventuale segno meno davanti a x
 if (y < 0)
 y = - y; //elimina l'eventuale segno meno davanti a y
 }
 }

 public Frazione per(Frazione f) {
 int n = this.num * f.num;
 int d = this.den * f.den;
 try {
 return new Frazione(n, d);
 } catch (FrazioneException e) {
 return null;
 }
 }

 public Frazione plus(Frazione f) throws FrazioneException {
 if (den != f.den)
 throw new FrazioneException("Impossibile sommare frazioni con denominatori diversi");
 else
 return per(new Frazione((num * f.den) + (f.num * den), den));
 }

 public Frazione minus(Frazione f) throws FrazioneException {
 if (den != f.den)
 throw new FrazioneException("Impossibile sottrarre frazioni con denominatori diversi");
 else
 return per(new Frazione((num * f.den) - (f.num * den), den));
 }

 public Frazione multiply(Frazione f) throws FrazioneException {
 if (den == 0 || f.den == 0)
 throw new FrazioneException("Impossibile moltiplicare frazioni con denominatori diversi da zero");
 else
 return per(new Frazione(num * f.num, den * f.den));
 }

 public Frazione divide(Frazione f) throws FrazioneException {
 if (den == 0 || f.den == 0)
 throw new FrazioneException("Impossibile dividere frazioni con denominatori diversi da zero");
 else
 return per(new Frazione(num * f.den, den * f.num));
 }

 public int compareTo(Frazione o) {
 if (den == o.den)
 return num - o.num;
 else
 return den - o.den;
 }
}
```

```
int m = mcd(x, y);
if (negativo)
 num = - x / m; //il segno meno viene memorizzato al numeratore
else
 num = x / m;
den = y / m;
}
}

public Frazione(int x) {
 this.num = x;
 this.den = 1;
}

public Frazione piu(Frazione f) {
 int n = this.num * f.den + this.den * f.num;
 int d = this.den * f.den;
 try {
 return new Frazione(n, d);
 } catch (FrazioneException e) {
 return null;
 }
}

public Frazione meno(Frazione f) {
 int n = this.num * f.den - this.den * f.num;
 int d = this.den * f.den;
 try {
 return new Frazione(n, d);
 } catch (FrazioneException e) {
 return null;
 }
}

public Frazione per(Frazione f) {
 int n = this.num * f.num;
 int d = this.den * f.den;
 try {
 return new Frazione(n, d);
 } catch (FrazioneException e) {
 return null;
 }
}
```

```
}

public Frazione diviso(Frazione f) throws FrazioneException {
 int n = this.num * f.den;
 int d = this.den * f.num;
 return new Frazione(n, d);
}

public boolean equals(Object altro) {
 if (altro instanceof Frazione) {
 Frazione a = (Frazione) altro;
 return equals(a);
 } else
 return false;
}

public boolean equals(Frazione f) {
 return this.num == f.num && this.den == f.den;
}

public boolean isMinore(Frazione f) {
 Frazione g = this.meno(f);
 if (g.num < 0)
 return true;
 else
 return false;
}

public boolean isMaggiore(Frazione f) {
 Frazione g = this.meno(f);
 if (g.num > 0)
 return true;
 else
 return false;
}

public String toString() {
 if (den == 0)
 return "impossibile";
 else if (den == 1)
 return String.valueOf(num);
 else
```

```
 return num + "/" + den;
 }

 public int getNumeratore() {
 return num;
 }

 public int getDenominatore() {
 return den;
 }

 public int compareTo(Frazione altra) {
 if (this.equals(altra))
 return 0;
 else if (this.isMinore(altra))
 return -1;
 else
 return 1;
 }

 private static int mcd(int a,int b) {
 int resto;
 do {
 resto = a % b;
 a = b;
 b = resto;
 } while (resto != 0);
 return a;
 }
}
```

Le versioni della classe Prova in cui viene intercettata `FrazioneException` possono essere compilate ed eseguite senza problemi. Consideriamo invece la prima versione della classe, cioè quella senza il trattamento delle eccezioni:

```
import prog.io.*;

class Prova {
 public static void main(String[] args) {
 ConsoleInputManager in = new ConsoleInputManager();
 ConsoleOutputManager out = new ConsoleOutputManager();

 int n = in.readInt("Numeratore prima frazione? ");
 int d = in.readInt("Denominatore prima frazione? ");
```

```

Frazione f1 = new Frazione(n, d);
out.println("E' stata inserita la frazione " + f1.toString());

n = in.readInt("Numeratore seconda frazione? ");
d = in.readInt("Denominatore seconda frazione? ");
Frazione f2 = new Frazione(n, d);
out.println("E' stata inserita la frazione " + f2.toString());

Frazione f3 = f1.diviso(f2);
out.print("Il quoziente di " + f1.toString() + " e " +
 f2.toString());
out.println(" è la frazione " + f3.toString());
}
}

```

Che cosa succede se tentiamo di compilarla? Il metodo `main` contiene due chiamate al costruttore con due argomenti e una chiamata al metodo `diviso`. Queste operazioni, come indicato nell'intestazione del costruttore e del metodo, possono sollevare l'eccezione controllata `FrazioneException`. Dunque `main` deve trattare questa eccezione. In questo caso, poiché il codice di `main` non prevede il trattamento dell'eccezione, il compilatore segnala errore:

`Prova.java:10: unreported exception FrazioneException;`

`must be caught or declared to be thrown`

`Frazione f1 = new Frazione(n, d);`

`^`

`Prova.java:15: unreported exception FrazioneException;`

`must be caught or declared to be thrown`

`Frazione f2 = new Frazione(n, d);`

`^`

`Prova.java:18: unreported exception FrazioneException;`

`must be caught or declared to be thrown`

`Frazione f3 = f1.diviso(f2);`

`^`

**3 errors**

Se la scelta per cui vogliamo optare è quella di rinviare l'eccezione alla Java Virtual Machine senza intercettarla in `main`, possiamo introdurre un'apposita clausola `throws` nell'intestazione di `main`:

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

## Documentazione delle eccezioni

Dato che le eccezioni sollevate dai metodi e dai costruttori costituiscono un'informazione fondamentale per l'utilizzatore della classe, anch'esse devono essere documentate. `javadoc` provvede

a inserire automaticamente nella documentazione l'elenco delle eccezioni delegate da un metodo senza, ovviamente, fornirne una descrizione. Per documentare le eccezioni delegate, javadoc prevede il seguente marcatore:

- `@throws nome_classe descrizione`

Viene utilizzato per documentare le eccezioni delegate da un metodo o da un costruttore. *nome\_classe* è il nome dell'eccezione delegata mentre *descrizione* è un testo che descrive l'eccezione e i casi in cui essa si verifica. Il testo può estendersi su più righe successive.

A titolo d'esempio riportiamo il commento di documentazione del costruttore con due argomenti della classe **Frazione**.

```
/**
 * Costruisce una nuova frazione il cui valore è il rapporto fra il
 * primo argomento e il secondo argomento; il denominatore
 * dev'essere diverso da zero, in caso contrario viene sollevata
 * un'eccezione controllata di tipo {@link FrazioneException}.
 * @param x numeratore.
 * @param y denominatore.
 * @throws FrazioneException nel caso in cui il secondo argomento
 * sia uguale a zero.
 */
public Frazione(int x, int y) throws FrazioneException {
 ...
}
```

## 11.9 Perché le eccezioni controllate?

Gli ultimi esempi, in cui abbiamo definito **FrazioneException** come controllata, evidenziano come il programmatore sia molto più vincolato quando sono in gioco eccezioni controllate: ogni volta che si scrive codice in cui può essere sollevata un'eccezione controllata è obbligatorio trattarla, intercettandola o delegandola. Ciò può apparire oneroso e fastidioso, ma è in realtà estremamente utile, in quanto impedisce al programmatore di "dimenticarsi" degli eventi rappresentati dalle eccezioni controllate, costringendolo a scegliere come gestirli.

Quando si introduce una nuova eccezione è importante decidere se definirla come controllata oppure no. Una scelta estrema potrebbe essere quella di definire tutte le eccezioni come controllate. Questo renderebbe però estremamente pesante la scrittura del codice (si pensi, ad esempio, a come diventerebbe noioso scrivere il codice se **NullPointerException** fosse controllata) e nel contempo non permetterebbe di evidenziare gli eventi realmente "eccezionali", e pertanto critici, rispetto a quelli evitabili con un'appropriata scrittura del codice.

Un criterio di massima per la scelta tra eccezioni controllate e non controllate è il seguente.

- Le anomalie legate a *eventi esterni* al programma vengono descritte da eccezioni controllate. Come esempio si consideri un'applicazione che deve accedere a una pagina web remota e alla mancanza o alla caduta della connessione di rete.

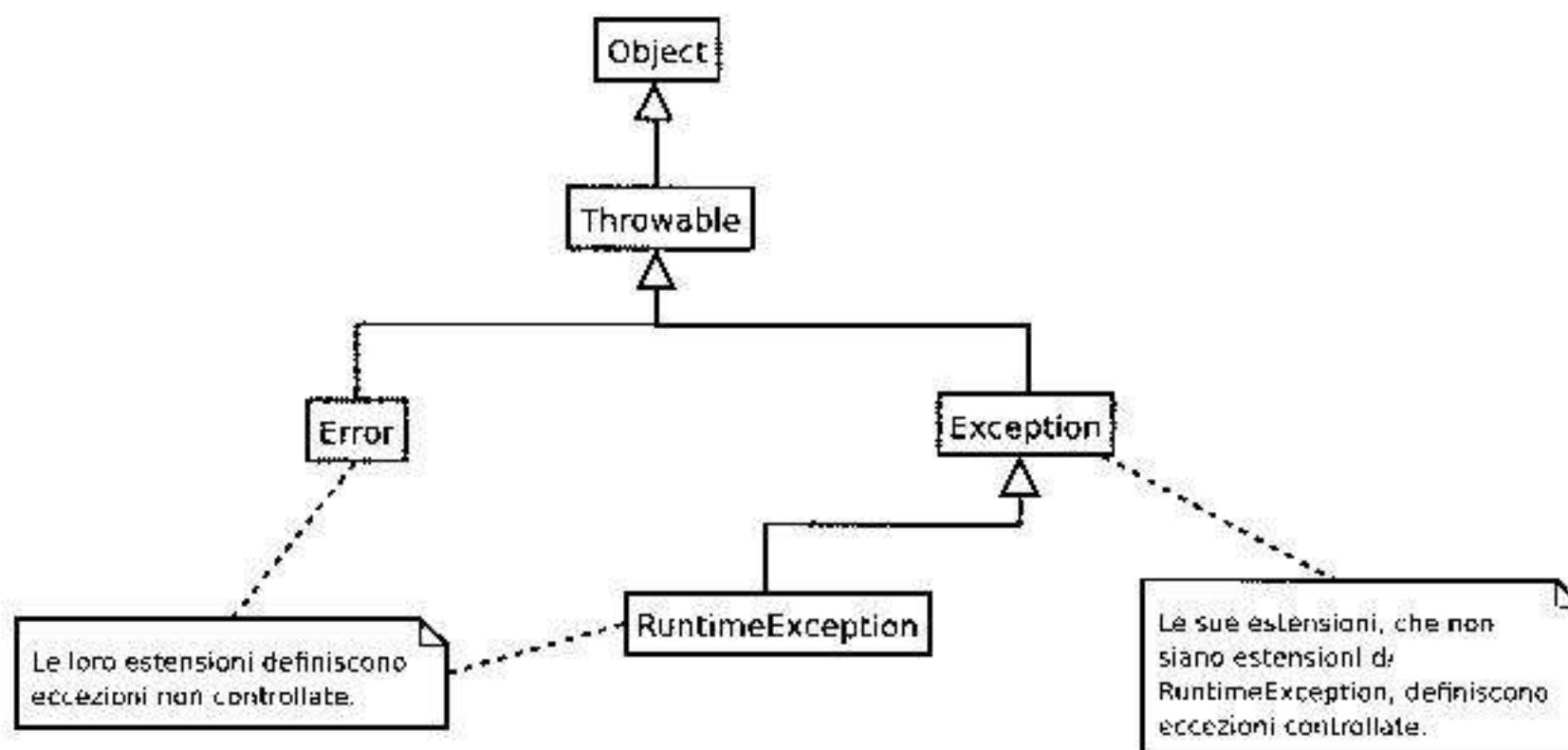


Figura 11.2 Gerarchia degli errori.

- Le anomalie legate a *eventi interni* al programma, cioè le anomalie che potrebbero essere evitate scrivendo il codice in maniera differente, vengono descritte da eccezioni non controllate. Ad esempio una divisione per zero o l'accesso a una posizione inesistente in un array potrebbero essere evitati inserendo nel codice opportuni controlli.

Nella libreria standard di Java tutte le eccezioni relative all'input/output sono controllate, in quanto legate a eventi esterni al programma, e dunque non prevenibili scrivendo il codice in maniera diversa. Nelle librerie per l'input/output che abbiamo fornito con il testo abbiamo scelto di utilizzare eccezioni non controllate per l'input/output solo per permetterne l'utilizzo prima di introdurre i concetti relativi alle eccezioni.<sup>10</sup>

## 11.10 Gli errori

Oltre alle eccezioni, ci sono altri oggetti “sollevabili” che prendono il nome di *errori*. Nella gerarchia essi sono rappresentati dalla sottoclasse Error di Throwable e dalle sue estensioni, come indicato nella Figura 11.2.

Anche gli errori possono essere trattati come eccezioni non controllate, ma, a differenza di esse, in genere definiscono situazioni irreparabili. Alcuni esempi sono OutOfMemoryError, che indica l'esaurimento della memoria disponibile, o NoClassDefFoundError, che indica che la Java Virtual Machine non riesce a trovare il bytecode di una classe da caricare per l'esecuzione.

<sup>10</sup> In base al criterio precedente, è opportuno definire l'eccezione FrazioneException come *non controllata*. L'esempio che abbiamo presentato, dove essa è controllata, aveva come scopo quello di mostrare la differenza di utilizzo tra eccezioni controllate e non.

## 11.11 La clausola finally

Nell'istruzione try-catch, dopo la parte try e le parti catch è possibile inserire un'ulteriore parte di codice, introdotta dalla parola riservata `finally`. La parte di codice introdotta da `finally` sarà comunque eseguita, sia che venga completato il blocco try senza sollevare l'eccezione sia che venga eseguito uno dei blocchi catch. Inoltre le istruzioni indicate nel blocco `finally` saranno eseguite anche qualora all'interno di try o di catch il normale flusso di esecuzione sia interrotto da istruzioni come `return`, `break` e `continue`.

Presenteremo ora un esempio che evidenzia il comportamento di questa clausola. Per una trattazione più approfondita del suo uso e della sua utilità rimandiamo ai manuali di riferimento del linguaggio.

Presentiamo una classe `Test` in cui viene calcolato il quoziente della divisione di due numeri interi inseriti dall'utente. Il metodo `main` si occupa della lettura dei numeri e della comunicazione del risultato, delegando il calcolo del quoziente a un altro metodo denominato `tentaDivisione`. Per semplicità, qualora non si possa effettuare la divisione, il metodo `tentaDivisione` restituisce 0. Il metodo `tentaDivisione` è basato su un costrutto try-catch-finally in cui, oltre alle operazioni richieste, vengono visualizzati messaggi utili a evidenziare il flusso dell'esecuzione. Il codice della classe `Test` è il seguente:

```
import prog.io.*;

class Test {
 private static ConsoleInputManager in = new ConsoleInputManager();
 private static ConsoleOutputManager out = new ConsoleOutputManager();

 public static void main(String[] args) {
 int x = in.readInt("Dividendo? ");
 int y = in.readInt("Divisore? ");
 int z = tentaDivisione(x, y);
 out.println("Il metodo tentaDivisione ha restituito " + z);
 }

 private static int tentaDivisione(int dividendo, int divisore) {
 try {
 out.println("tentaDivisione: inizio esecuzione blocco try");
 return dividendo / divisore;
 } catch (ArithmetcException e) {
 out.println("Eccezione intercettata: esecuzione blocco catch");
 return 0;
 } finally {
 out.println("tentaDivisione: esecuzione blocco finally");
 }
 }
}
```

}

Mostriamo ora due esempi di esecuzione. Nel primo inseriamo un divisore diverso da zero:

```
Dividendo? 10
Divisore? 4
tentaDivisione: inizio esecuzione blocco try
tentaDivisione: esecuzione blocco finally
Il metodo tentaDivisione ha restituito 2
```

In questo caso la divisione viene effettuata correttamente. Dunque il blocco `catch` non è eseguito. Nel blocco `try` viene restituito al chiamante il risultato della divisione; prima del rientro effettivo dal metodo viene però eseguito il blocco `finally`, come evidenziato dall'output prodotto.

Presentiamo ora un esempio in cui inseriamo come divisore zero:

```
Dividendo? 10
Divisore? 0
tentaDivisione: inizio esecuzione blocco try
Eccezione intercattata: esecuzione blocco catch
tentaDivisione: esecuzione blocco finally
Il metodo tentaDivisione ha restituito 0
```

In questo caso l'operazione di divisione solleva un'eccezione intercattata dal blocco `catch`, come evidenziato dal relativo messaggio. Anche qui, prima del rientro effettivo dal metodo, viene eseguito il blocco `finally`.

## 11.12 Ridefinizione di metodi ed eccezioni

Nel Capitolo 8 abbiamo mostrato come nelle sottoclassi sia possibile ridefinire i metodi delle superclassi. In particolare, per ridefinire un metodo di una superclasse occorre implementare nella sottoclasse un metodo con lo stesso prototipo.

Ridefinendo un metodo è necessario tenere conto anche delle eccezioni controllate che esso può delegare con la clausola `throws`: l'insieme di eccezioni controllate che il metodo della sottoclasse dichiara di delegare non può essere più ampio dell'insieme di eccezioni controllate delegate dal metodo della superclasse che viene ridefinito.

Consideriamo ad esempio le seguenti eccezioni:

```
class Eccezione0 extends Exception {
}

class SottoEccezione1 extends Eccezione0 {
```

```
class SottoEccezione2 extends Eccezione0 {
}
```

e le classi:

```
class A {

 public void f(int i) throws Eccezione0 {
 throw new Eccezione0();
 }
}
```

```
class E extends A {

 public void f(int i) throws Eccezione0 {
 ...
 }
}
```

```
class D extends A {

 public void f(int i) {
 ...
 }
}
```

```
class B extends A {

 public void f(int i) throws SottoEccezione1, SottoEccezione2 {
 if (i < 0)
 throw new SottoEccezione1();
 else
 throw new SottoEccezione2();
 }
}
```

Il metodo `f` della classe `A` viene ridefinito nelle sottoclassi `B`, `D` ed `E`. Nei tre casi la ridefinizione del metodo è corretta, in quanto l'insieme di eccezioni del metodo della sottoclasse non è più ampio dell'insieme di eccezioni del metodo ridefinito della superclasse. In particolare:

- il metodo `f` di `E` delega `Eccezione0` come il metodo `f` della superclasse `A`;
- il metodo `f` di `D` non delega eccezioni;

- il metodo `f` di `B` delega eccezioni di tipo `SottoEccezione1` e `SottoEccezione2`, sottoclassi di `Eccezione0`.

Consideriamo ora la seguente classe `C`, estensione di `B`:

```
class C extends B {

 public void f(int i) throws Eccezione0 {
 throw new Eccezione0();
 }
}
```

Il tipo delle eccezioni delegate dal metodo `f` di `C` è più ampio di quello delle eccezioni delegate dal metodo della superclasse `B`. In questa situazione il codice della classe non viene compilato; in particolare il compilatore fornisce il seguente messaggio:

```
C.java:3: f(int) in C cannot override f(int) in B; overridden method
does not throw Eccezione0
 public void f(int i) throws Eccezione0 {
 ^
1 error
```

Il vincolo presentato, relativo alle eccezioni delegate dai metodi ridefiniti, garantisce che il gestore di un'eccezione sollevata dal metodo `f` di un'istanza di `A` operi correttamente anche per tutti gli oggetti che sono istanze di sottoclassi di `A`. Consideriamo, ad esempio, il seguente blocco `try-catch`:

```
try {
 o.f(1);
} catch (Eccezione0 e) {
 ...
}
```

Se il riferimento `o` è stato dichiarato di tipo `A`, l'oggetto riferito da `o` dev'essere un'istanza di `A` o di una sua sottoclasse. Nell'intestazione del metodo `f` di `A` è dichiarato che le uniche eccezioni controllate che esso può delegare sono istanze di `Eccezione0` (o di sue sottoclassi). Il fatto che ridefinendo il metodo nelle sottoclassi non sia possibile ampliare l'insieme di eccezioni controllate delegate garantisce che il blocco `try-catch`, scritto sopra, intercetti correttamente le eccezioni sollevate dal metodo `f` anche quando l'oggetto riferito da `o` è un'istanza di una sottoclasse di `A`.

È ovviamente possibile differenziare il comportamento del gestore a seconda dell'eccezione sollevata così come segue:

```
try {
 o.f(1);
} catch (SottoEccezione1 e) {
```

```

 ...
} catch (SottoEccezione2 e) {
 ...
} catch (Eccezione0 e) {
 ...
}

```

In questo caso il blocco `catch` eseguito può dipendere dal tipo dell'oggetto referenziato da `o` in base all'eccezione sollevata.

## 11.13 Metodi astratti ed eccezioni

Come abbiamo visto, i metodi astratti definibili nelle classi astratte (Capitolo 8) e nelle interfacce (Capitolo 9) sono metodi di cui viene fornito il prototipo ma non l'implementazione. Specificando l'intestazione di un metodo astratto è possibile indicare le eccezioni che tale metodo potrà delegare. Implementando un metodo astratto dovremo rispettare quanto dichiarato nel metodo: anche in questo caso, come nel caso della ridefinizione dei metodi, non potremo far delegare ai metodi effettivamente implementati tipi di eccezioni più ampi rispetto a quanto indicato nel metodo astratto.

Considerate le eccezioni `Eccezione0`, `SottoEccezione1` e `SottoEccezione2` definite nel paragrafo precedente. Quello che segue è un esempio di implementazione di un metodo astratto di una classe astratta:

```

abstract class A {
 abstract public void f(int i) throws Eccezione0;
}

class B extends A {

 public void f(int i) throws SottoEccezione1, SottoEccezione2 {
 if(i<0)
 throw new SottoEccezione1();
 else
 throw new SottoEccezione2();
 }
}

```

Il seguente è invece un esempio di implementazione di un'interfaccia con un metodo astratto che delega alcune eccezioni:

```

abstract interface I {
 public void f(int i) throws Eccezione0;
}

```

}

```
class A implements I {

 public void f(int i) throws SottoEccezione1, SottoEccezione2 {
 if (i < 0)
 throw new SottoEccezione1();
 else
 throw new SottoEccezione2();
 }
}
```

# **Parte III**

# **Argomenti avanzati**

# Strutture dati dinamiche

Durante l'esecuzione dei programmi è necessario memorizzare e organizzare i dati utilizzati in apposite strutture, dette *strutture dati*. Alcuni semplici esempi di strutture dati sono le pile e gli insiemi. La scelta delle strutture dati da usare in un programma dipende principalmente dalle operazioni che dovranno essere effettuate su di esse. Ad esempio in alcune strutture le operazioni di inserimento di un elemento sono estremamente efficienti in termini di tempo, mentre le operazioni di ricerca risultano dispendiose. Altre strutture hanno invece caratteristiche opposte. Le librerie standard di Java forniscono svariate classi che implementano le principali strutture dati. Ad esempio la classe `Stack`, utilizzata nel Capitolo 11 per costruire la calcolatrice in notazione postfissa, implementa le strutture a pila.

In questo capitolo introdurremo alcune strutture dati fondamentali e mostreremo la loro implementazione in linguaggio Java. In particolare concentreremo la nostra attenzione sulle strutture dinamiche, cioè su quelle strutture la cui organizzazione evolve dinamicamente durante l'esecuzione in base ai dati che devono essere memorizzati. Ad esempio, mentre un array è statico in quanto la sua struttura viene definita al momento della creazione e non è modificabile, una pila può essere espansa o contratta dinamicamente durante l'esecuzione aggiungendo o eliminando elementi. Il capitolo non esaurisce l'argomento, ma ne costituisce solo una breve introduzione. Noi ci soffermeremo principalmente sulla manipolazione dei tipi definiti ricorsivamente, sviluppando opportuni metodi di esempio. Le classi che presentiamo devono essere viste come semplici "raccoglitori" di tali metodi. La progettazione di classi per le strutture dinamiche è un argomento che richiede notevoli approfondimenti e per il quale rimandiamo ai numerosi testi a esso dedicati.

### 12.1 Implementazione di strutture a pila

Come primo esempio implementiamo una parte della classe generica `Stack<E>` definita nella libreria `java.util`. In particolare forniremo i seguenti metodi fondamentali:

- `public void push(E o)`

Aggiunge in cima alla pila che esegue il metodo l'oggetto fornito tramite il parametro.<sup>1</sup>

---

<sup>1</sup> Ricordiamo che il metodo `push` della classe `java.util.Stack` restituisce, in realtà, un riferimento all'oggetto

- **public E pop()**  
Restituisce un riferimento all'oggetto che si trova in cima alla pila che esegue il metodo eliminandolo dalla pila stessa. Se la pila è vuota, il metodo solleva una `EmptyStackException`.
- **public boolean empty()**  
Restituisce `true` se e solo se la pila che esegue il metodo è vuota.

La classe disporrà inoltre di un costruttore privo di argomenti che crea un oggetto che rappresenta una pila vuota.

Insieme con la classe `Stack` definiamo anche la classe per l'eccezione (non controllata) che può essere sollevata dal metodo `pop`. La classe è estremamente semplice:

```
public class EmptyStackException extends RuntimeException {
}
```

## Implementazione di pile mediante array

Un'implementazione naturale della classe `Stack` può essere basata sugli array. Una pila di oggetti viene rappresentata da un array nel quale ogni posizione è utilizzata per memorizzare un elemento della pila. In particolare la posizione 0 dell'array viene utilizzata per memorizzare l'oggetto che si trova più in basso nella pila. Per sapere dove si trova la cima della pila o, meglio, la prima posizione dell'array libera dopo la fine della pila, si utilizza un indice intero. Se questo indice contiene ad esempio 3, significa che la pila è formata dai 3 oggetti memorizzati nelle posizioni 0, 1 e 2 dell'array.

Prima di fornire un'implementazione della classe generica `Stack`, in questo paragrafo presentiamo, a titolo di esempio, l'implementazione di una classe (non generica) `Pila`, per rappresentare pile di oggetti (cioè di elementi di tipo `Object`).

La classe viene realizzata con due variabili di istanza: l'array `dati` (i cui elementi sono di tipo `Object`) e l'indice `top` che rappresenta, di volta in volta, la prima posizione libera. C'è inoltre un campo statico `SIZE` utilizzato per definire la lunghezza dell'array. In questa versione di esempio, non implementiamo la classe come generica.

Il costruttore della classe ha il compito di creare una pila vuota: pertanto costruisce l'array e assegna al campo `top` il valore 0 per indicare che questa è la prima posizione libera:

```
public Pila() {
 dati = new Object[SIZE];
 top = 0;
}
```

Il metodo `push` non fa altro che assegnare il riferimento ricevuto tramite il parametro alla posizione indicata da `top` e incrementa `top`.

---

inserito nello stack.

```
public void push(Object o) {
 dati[top++] = o;
}
```

Il metodo `pop` svolge operazioni del tutto simmetriche. Tuttavia, nel caso di pila vuota (cioè quando `top` contiene zero), anziché restituire il risultato, il metodo deve sollevare un'eccezione:

```
public Object pop() {
 if (top == 0)
 throw new EmptyStackException();
 else
 return dati[--top];
}
```

Il metodo `empty` controlla infine il valore di `top`. Ecco il codice completo della classe:

```
public class Pila {
 //CAMPI
 private static final int SIZE = 10;
 private Object[] dati;
 private int top; //indica la prima posizione dell'array

 //COSTRUTTORI
 public Pila() {
 dati = new Object[SIZE];
 top = 0;
 }

 //METODI
 public void push(Object o) {
 dati[top++] = o;
 }

 public Object pop() {
 if (top == 0)
 throw new EmptyStackException();
 else
 return dati[--top];
 }

 public boolean empty() {
 return top == 0;
 }
}
```

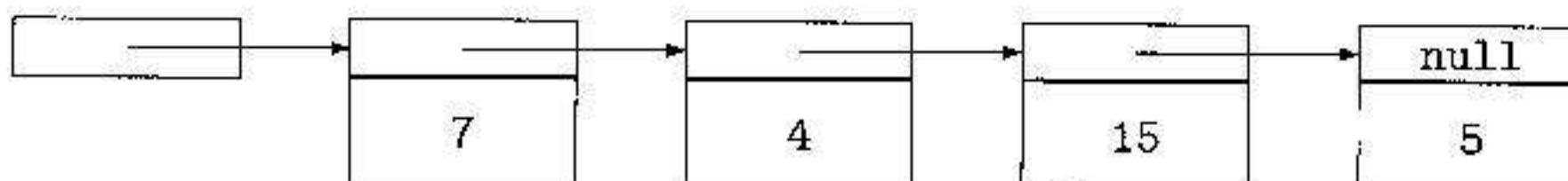
Questa implementazione delle strutture a pila è estremamente semplice. Il grosso svantaggio è che la dimensione massima della pila viene fissata a priori: se si tenta di eseguire il metodo `push` quando la pila è già piena, sarà sollevata una `ArrayIndexOutOfBoundsException`. Complicando un po' la struttura della classe, è possibile risolvere questo problema costruendo un nuovo array più grande.

Un secondo problema, nel definire la classe generica `Stack` a partire da questa implementazione di `Pila`, sta nell'uso degli array. Potremmo pensare di sostituire al tipo `Object`, il tipo parametro `E`. Il compilatore, tuttavia, non accetta l'invocazione del costruttore dell'array `new E[SIZE]`, all'interno del costruttore della classe. In generale, non è possibile costruire array di un tipo parametro.<sup>2</sup> Si dovrebbero utilizzare comunque un array di `Object` e un cast al tipo `E`, nel metodo `pop`, per restituire il risultato.

## Implementazione di pile mediante strutture dinamiche

Presentiamo ora una seconda implementazione delle pile basata su una struttura dati dinamica: anziché creare subito lo spazio per la pila (come nel caso dell'array), lo creiamo dinamicamente, man mano che risulta necessario. In particolare creiamo lo spazio per il nuovo elemento da aggiungere alla pila ogni volta che effettuiamo l'operazione `push`. Simmetricamente, quando si esegue un'operazione `pop`, lo spazio occupato dall'elemento prelevato dalla pila può essere rilasciato.

La struttura usata per rappresentare la pila è una *lista concatenata*, cioè un insieme di nodi, collegati tra loro mediante riferimenti. Ad esempio la pila contenente gli interi 5 15 4 7 (con 7 sulla cima) può essere rappresentata mediante la lista nella seguente figura (per comodità indichiamo direttamente i valori nei nodi; in realtà ogni nodo conterrà un riferimento all'oggetto, in questo caso di tipo `Integer`, contenente il valore memorizzato nel nodo):



La lista è costituita da quattro nodi con la medesima struttura: un campo per l'oggetto da memorizzare nel nodo e un'indicazione che permette di accedere al nodo successivo. Quest'ultima informazione può essere rappresentata tramite un riferimento (detto anche *puntatore*). C'è inoltre un riferimento alla testa, cioè al primo elemento della lista. In questa rappresentazione, il primo elemento della lista corrisponde alla cima della pila; nella lista ogni elemento è seguito dall'elemento che nella pila si trova sotto di esso. L'ultimo elemento della lista non ha successori e corrisponde all'elemento che si trova più in basso nella pila.

Un nodo è dunque un oggetto costituito da due campi:

- un campo che chiameremo `dato`, destinato a contenere il riferimento a ciò che si è interessati a memorizzare nel nodo;

<sup>2</sup> Questa limitazione è dovuta alla differente gestione dei tipi generici e degli array, cui abbiamo accennato nel Paragrafo 6.18.

- un campo `pros` che permette di accedere all'elemento successivo della lista, cioè all'elemento sottostante nella pila.

Si osservi che il campo `pros` è un riferimento a un oggetto dello stesso tipo del nodo. In altre parole, se chiamiamo `NodoStack` il tipo del nodo, allora `pros` è di tipo `NodoStack`. Possiamo definire la struttura di un nodo utilizzando la seguente classe (senza metodi e con il costruttore implicito privo di argomenti), nella quale indichiamo con `E` il tipo degli oggetti che vogliamo memorizzare nella pila:<sup>3</sup>

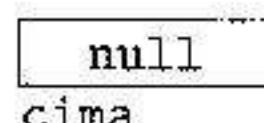
```
class NodoStack {
 E dato;
 NodoStack pros;
}
```

Si noti che la definizione precedente è ricorsiva, cioè la struttura delle istanze di `NodoStack` è definita in termini di se stessa.

Per definire la pila è sufficiente fornire un riferimento al nodo contenente l'elemento che si trova in cima (tale riferimento sarà `null` nel caso di pila vuota):

```
public class Stack<E> {
 private NodoStack cima;
 ...
}
```

Riscriviamo ora il costruttore e i metodi della classe `Stack`. La classe ha un costruttore, privo di argomenti, che costruisce una pila vuota. In questa rappresentazione la pila vuota, visualizzata nella figura seguente, si ottiene assegnando `null` al campo `cima`:



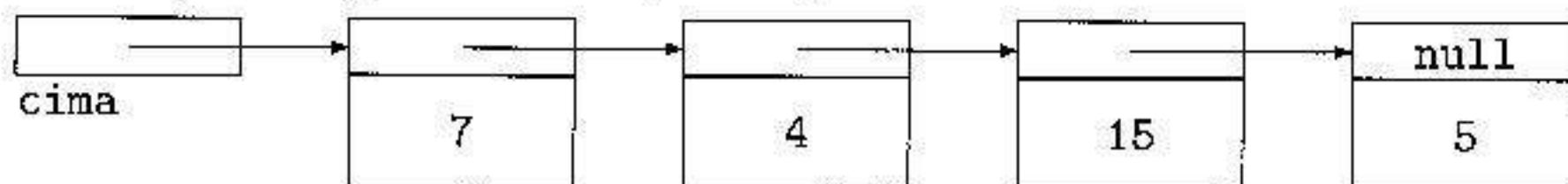
```
public Stack() {
 cima = null;
}
```

Possiamo scrivere subito il metodo `empty`, il cui compito è controllare se la pila è vuota. Il metodo deve esaminare il riferimento contenuto in `cima`:

```
public boolean empty() {
 return cima == null;
}
```

<sup>3</sup> Come vedremo più avanti, la classe `NodoStack` verrà collocata all'interno della classe `Stack`, di cui `E` è tipo parametro. Per questa ragione non è necessario indicare `E` come tipo parametro per `NodoStack`.

Sviluppiamo ora il metodo push. Ricordiamo che tale metodo riceve, tramite il parametro *o* di tipo E, il riferimento a un oggetto che dev'essere posto in cima alla pila o, in altre parole, all'inizio della lista. Supponiamo che la pila contenga già alcuni elementi. In particolare supponiamo che il contenuto sia quello rappresentato in questa figura:



Il nuovo elemento dovrà essere inserito *all'inizio* della lista. A tale scopo, le operazioni da effettuare sono:

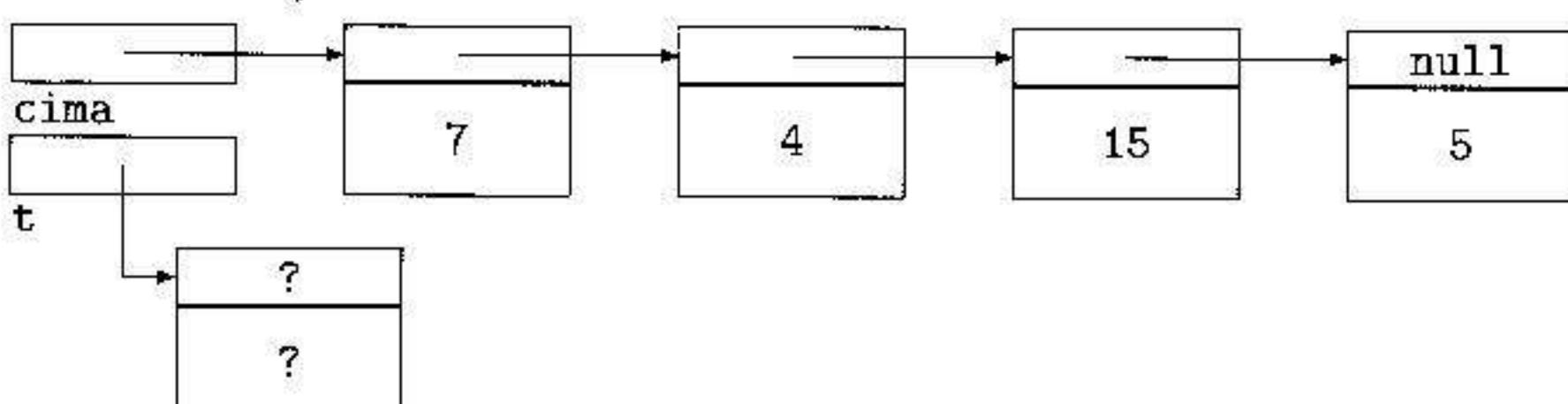
- creazione di un nuovo nodo
- copia del riferimento o nel campo dato del nuovo nodo
- inserimento del nuovo nodo all'inizio della lista.

Dichiariamo localmente al metodo push un riferimento *t* di tipo NodoStack, che utilizziamo per la creazione del nuovo nodo.

Per creare il nodo basta invocare il costruttore privo di argomenti di NodoStack scrivendo:

```
t = new NodoStack();
```

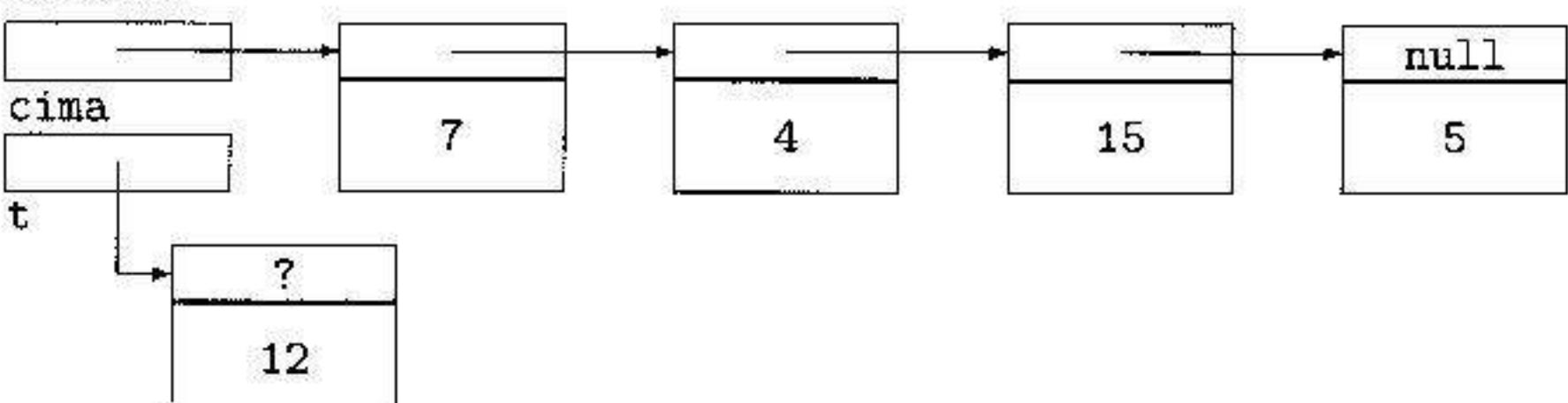
Dopo l'esecuzione di questa istruzione, la memoria conterrà:



(Nella figura abbiamo indicato con un punto interrogativo i contenuti dei campi del nuovo nodo, per ricordarci che dobbiamo assegnare loro i valori. In realtà tali campi, essendo di tipo riferimento, vengono inizializzati automaticamente a null dal costruttore.) Per assegnare il riferimento contenuto in *o* al campo dato del nuovo nodo è sufficiente scrivere:

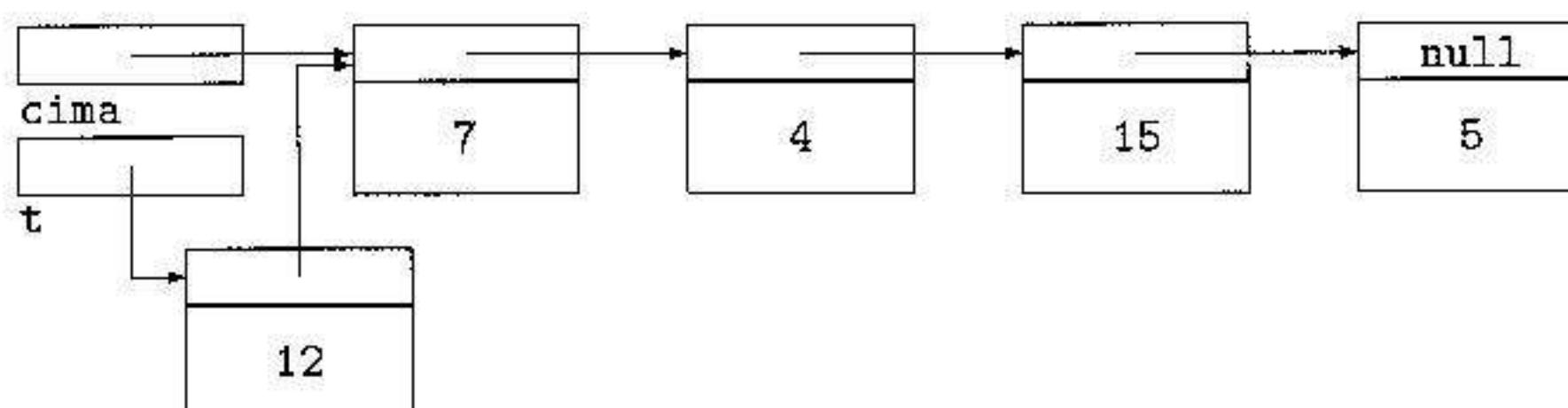
```
t.dato = o;
```

Supponendo che *o* si riferisca a un oggetto Integer contenente il valore 12, il contenuto della memoria diventa:

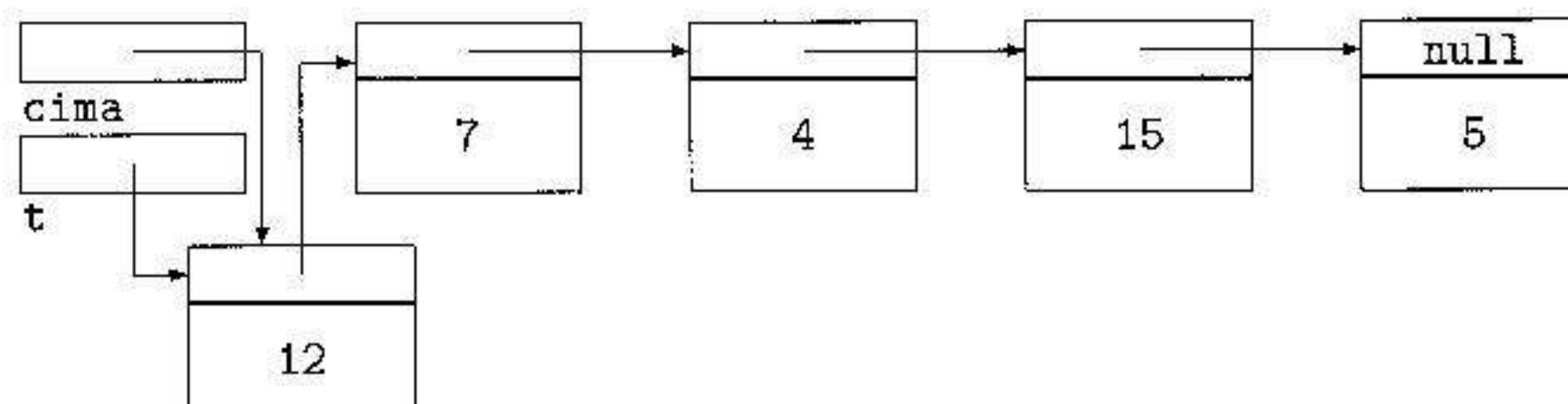


A questo punto occorre collegare il nuovo nodo alla lista già esistente. Quest'operazione viene attuata in due fasi:

- si fa puntare il campo `pros` del nuovo nodo all'inizio della lista esistente mediante l'assegnamento `t.pros = cima`:



- si fa puntare l'inizio della lista, cioè il riferimento `cima`, al nuovo nodo mediante l'assegnamento `cima = t`:



Osservando ora il contenuto della lista a partire da `cima` si ottiene la sequenza 12 7 4 15 5, che rappresenta appunto la pila letta a partire dalla cima. Il codice completo del metodo è dunque:

```
public void push(E o) {
 NodoStack t = new NodoStack();
 t.dato = o;
 t.pros = cima;
 cima = t;
}
```

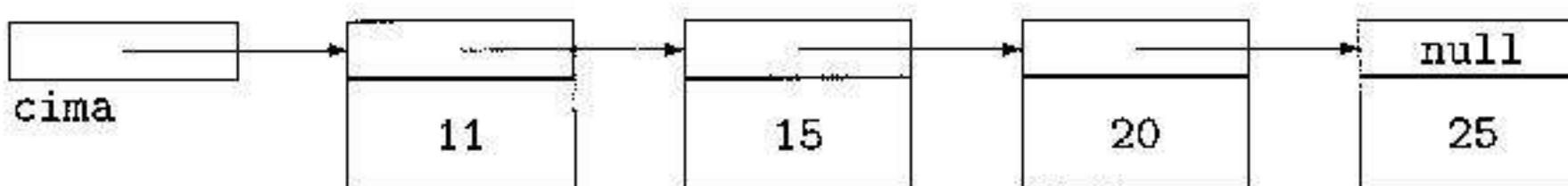
Scriviamo ora il metodo `pop`. Compito di questo metodo è eliminare dalla pila l'oggetto che si trova sulla cima, restituendo come risultato un riferimento a esso. Se la pila è vuota, il metodo deve sollevare l'eccezione `EmptyStackException`. In caso contrario il metodo effettua le seguenti operazioni:

- memorizza in una variabile `risultato` di tipo E il riferimento contenuto nel primo elemento della lista (cioè nel campo `dato`);
- elimina dalla lista il primo nodo (cioè l'elemento in cima alla pila);
- restituisce il riferimento contenuto in `risultato`.

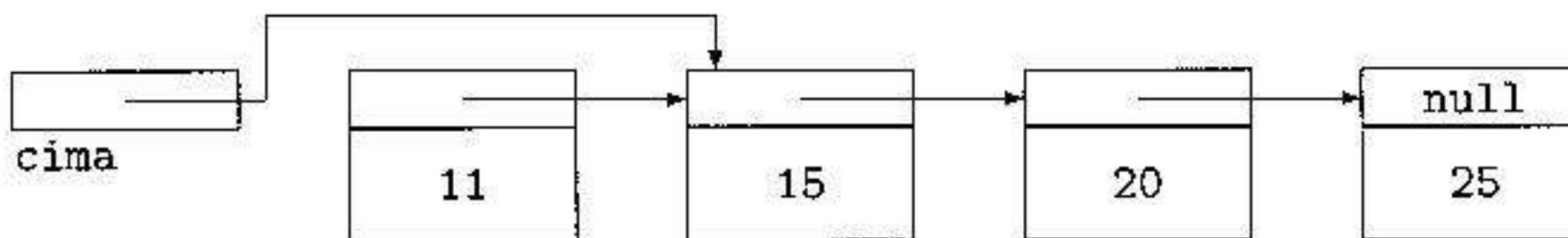
Per assegnare alla variabile `risultato` il contenuto del primo nodo, cioè del nodo riferito da `cima`, è sufficiente scrivere:

```
risultato = cima.dato;
```

Vediamo come eliminare il primo nodo. Supponete che la pila sia rappresentata dalla seguente lista:



In questo caso, per sganciare il nodo dalla lista basta far riferire `cima` al secondo nodo della lista. Un riferimento a tale nodo è presente nel campo `pros` dell'oggetto riferito da `cima`. In altre parole, per eliminare il nodo dalla lista si può utilizzare l'assegnamento `cima = cima.pros`:



Lo spazio occupato dal nodo sganciato dalla lista, non più accessibile, potrà essere poi recuperato dal garbage collector.

Ecco il codice completo del metodo `pop()`:

```

public E pop() {
 if (cima == null)
 throw new EmptyStackException();
 else {
 E risultato = cima.dato;
 cima = cima.pros;
 return risultato;
 }
}

```

La classe `NodoStack` è stata pensata esclusivamente per implementare `Stack`. Dunque l'unica classe interessata a utilizzare `NodoStack` è `Stack`. È pertanto opportuno dichiarare `NodoStack` come risorsa privata all'interno della classe `Stack` stessa. I campi di `NodoStack` `dato` e `pros` non sono preceduti da modificatori: risultano così accessibili direttamente dal codice della classe

Stack. Essi non sono tuttavia accessibili all'esterno della classe Stack, in quanto non lo è la classe NodoStack.<sup>4</sup>

Ecco il codice completo di questa nuova versione di Stack:

```
public class Stack<E> {
 private NodoStack cima;

 private class NodoStack {
 E dato;
 NodoStack pros;
 }

 public Stack() {
 cima = null;
 }

 public void push(E o) {
 NodoStack t = new NodoStack();
 t.dato = o;
 t.pros = cima;
 cima = t;
 }

 public E pop() {
 if (cima == null)
 throw new EmptyStackException();
 else {
 E risultato = cima.dato;
 cima = cima.pros;
 return risultato;
 }
 }

 public boolean empty() {
 return cima == null;
 }
}
```

<sup>4</sup> Questo è il primo punto nel testo in cui scriviamo una classe all'interno di un'altra. Avremmo anche potuto procedere nel modo usuale, mantenendo la classe NodoStack separata dalla classe Stack e scrivendola nello stesso file o in un file differente. Tuttavia, una soluzione di questo tipo non avrebbe molto significato, in quanto l'unico scopo per cui abbiamo scritto la classe NodoStack è quello di implementare Stack. Pertanto, porre NodoStack all'interno di Stack enfatizza il fatto che NodoStack è parte dell'implementazione di Stack. Inoltre, poiché il campo dato di NodoStack è del tipo parametro della classe Stack, dichiarando NodoStack al di fuori di Stack, sarebbe necessario renderla generica.

Una classe definita, come in questo esempio, all'interno di un'altra, viene detta *classe interna*; una classe statica definita all'interno di un'altra viene detta *classe innestata*. Non ci addentreremo ulteriormente nell'esame di questi aspetti, per i quali rimandiamo ai manuali del linguaggio.

۲

## Esercizi

- 12.1 Scrivete un'applicazione che legga una sequenza di numeri interi e la riscriva alla rovescia, cioè a partire dall'ultimo elemento inserito. Per memorizzare gli interi utilizzate una struttura a pila.

12.2 Scrivete un metodo che legga una stringa, costruisca una nuova stringa ottenuta rovescian-  
do la stringa data, e la visualizzi. Ad esempio, se l'utente inserisce `roma`, il metodo dovrà  
visualizzare `amor`. Scrivete il metodo in due versioni: la prima usando esclusivamente la  
classe `String`, la seconda ricorrendo a una struttura a pila in cui gli oggetti memorizzati  
rappresentano caratteri.

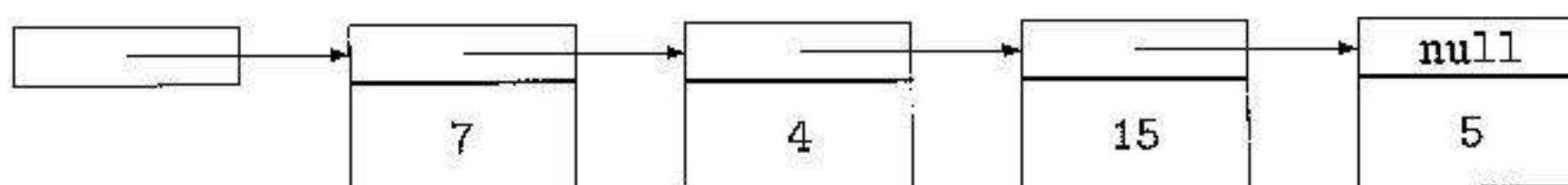
## 12.2 Le code

Nelle pile gli elementi vengono prelevati e inseriti sempre dallo stesso lato. In questo modo, il primo elemento a essere prelevato è sempre quello che è stato inserito per ultimo (struttura LIFO: *Last In First Out*).

Consideriamo ora le strutture a *coda* in cui, a differenza delle pile, gli elementi vengono prelevati da un lato e inseriti dal lato opposto. In una coda il primo elemento che può essere prelevato è dunque quello che è stato inserito per primo (struttura FIFO: *First In First Out*). Pertanto gli elementi vengono prelevati nello stesso ordine con cui sono stati inseriti. Si pensi, come esempio, a una fila di persone in coda davanti allo sportello di un ufficio postale.

Anche le code possono essere realizzate sia con array sia con liste. Presenteremo solo la realizzazione di una coda di oggetti mediante liste.

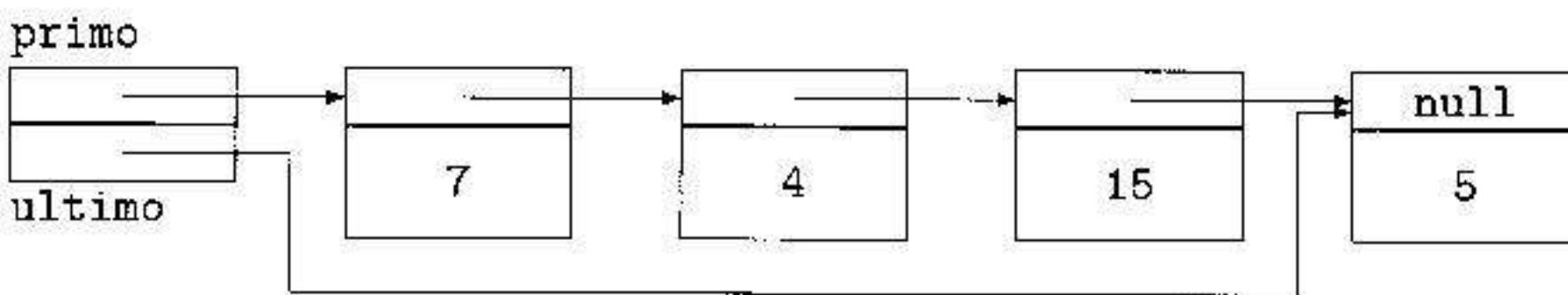
Una coda può essere realizzata con una lista in cui gli inserimenti vengono effettuati *alla fine* di essa. Considerate ad esempio la seguente lista:



Per inserire un nuovo nodo *alla fine* della lista occorre:

- creare un nuovo nodo;
  - percorrere l'intera lista fino a raggiungere l'ultimo nodo;
  - collegare il nuovo nodo alla lista facendo puntare a esso il riferimento contenuto nell'ultimo nodo.

In termini di tempo questa tecnica è molto dispendiosa. Ogni volta che si vuole inserire un nuovo elemento è infatti necessario scandire l'intera lista. Per evitare di effettuare questo processo, si può rappresentare la struttura con due riferimenti, uno al primo, l'altro all'ultimo elemento:



La coda vuota viene invece rappresentata ponendo a `null` ambedue i riferimenti `primo` e `ultimo`.

Vediamo ora come realizzare una classe generica `Coda`. Analogamente a quanto fatto per la classe `Stack`, definiamo una classe per descrivere i nodi della coda:

```

class NodoCoda {
 E dato;
 NodoCoda pros;
}

```

Porremo questa classe all'interno della classe `Coda` dichiarandola come `private` e `static`. La classe `Coda` conterrà due campi di tipo `NodoCoda`: i riferimenti al primo e all'ultimo elemento della coda.

```

public class Coda<E> {
 private NodoCoda primo, ultimo;

 private static class NodoCoda {
 E dato;
 NodoCoda pros;
 }

 ...costruttori e metodi...
}

```

Il costruttore di `Coda` si occupa di costruire una coda vuota. A questo fine è sufficiente assegnare `null` ai campi `primo` e `ultimo` (in realtà potremmo evitare di scrivere il costruttore, in quanto questa è già l'inizializzazione di default per i campi contenenti riferimenti):

```

public Coda() {
 primo = ultimo = null;
}

```

Scriviamo ora un metodo `aggiungi` per l'inserimento di un nuovo elemento. Il metodo avrà la seguente intestazione:

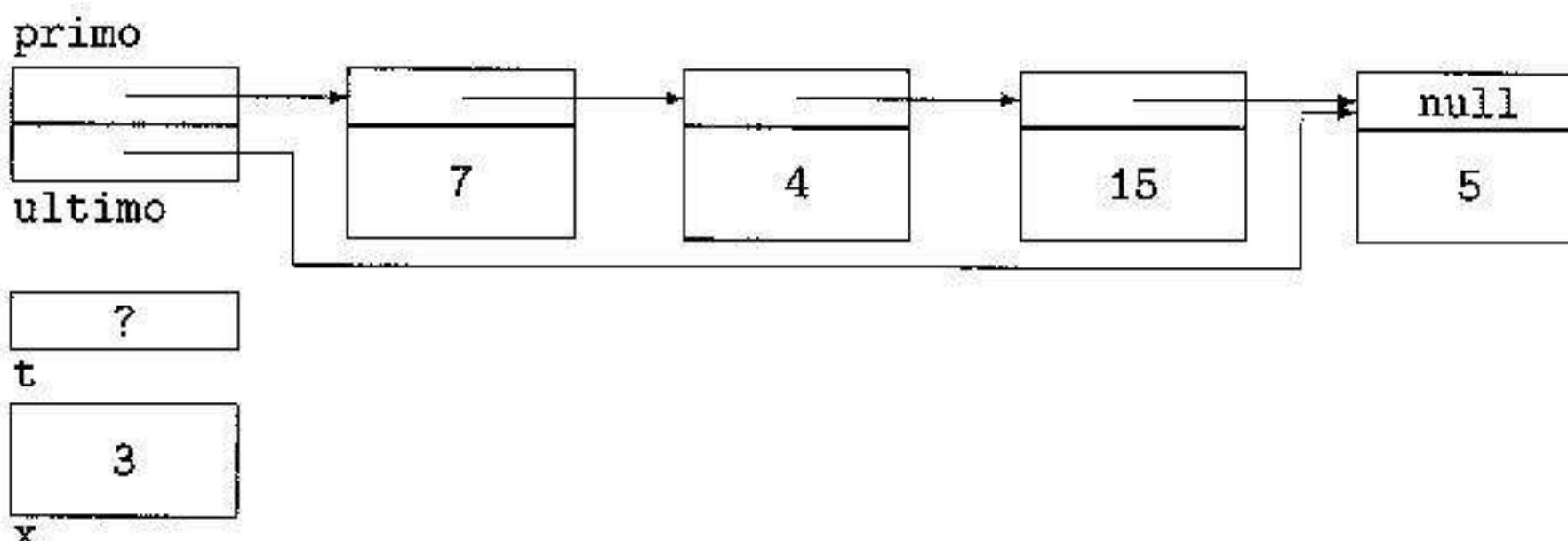
```

public void aggiungi(E x)

```

dove il parametro **x** contiene un riferimento all'oggetto da inserire nella coda. Facciamo uso di un riferimento ausiliario **t** di tipo **NodoCoda**, utile per la creazione del nuovo elemento.

Supponiamo che la situazione prima dell'inserimento sia quella rappresentata nella seguente figura (come nel caso delle pile, per semplicità rappresentiamo i campi **dato** dei nodi con interi; in realtà, nel campo **dato** di ciascun nodo viene memorizzato il riferimento all'oggetto da memorizzare; analogamente ricordiamo che il parametro **x** contiene un riferimento all'oggetto da inserire, ma nelle figure indichiamo il contenuto dell'oggetto direttamente in un riquadro di nome **x**):



Per inserire nella coda un nuovo nodo per l'oggetto riferito da **x** effettuiamo le seguenti operazioni.

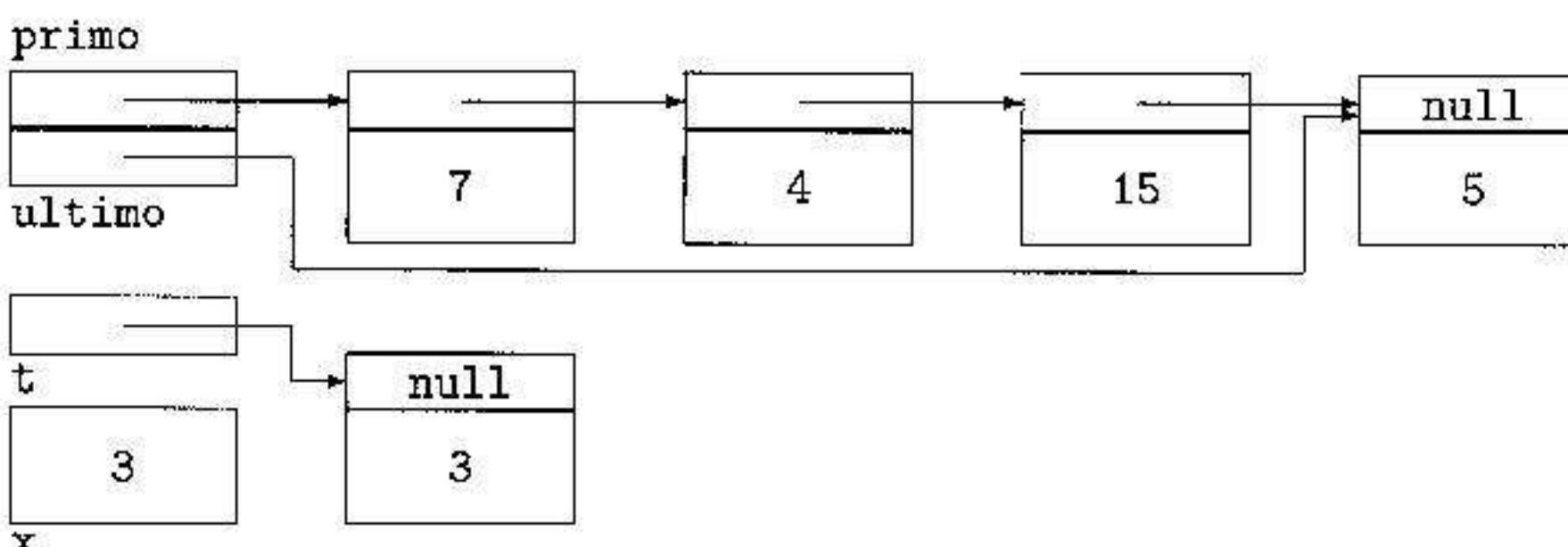
- Creazione del nuovo nodo e registrazione delle informazioni:

```

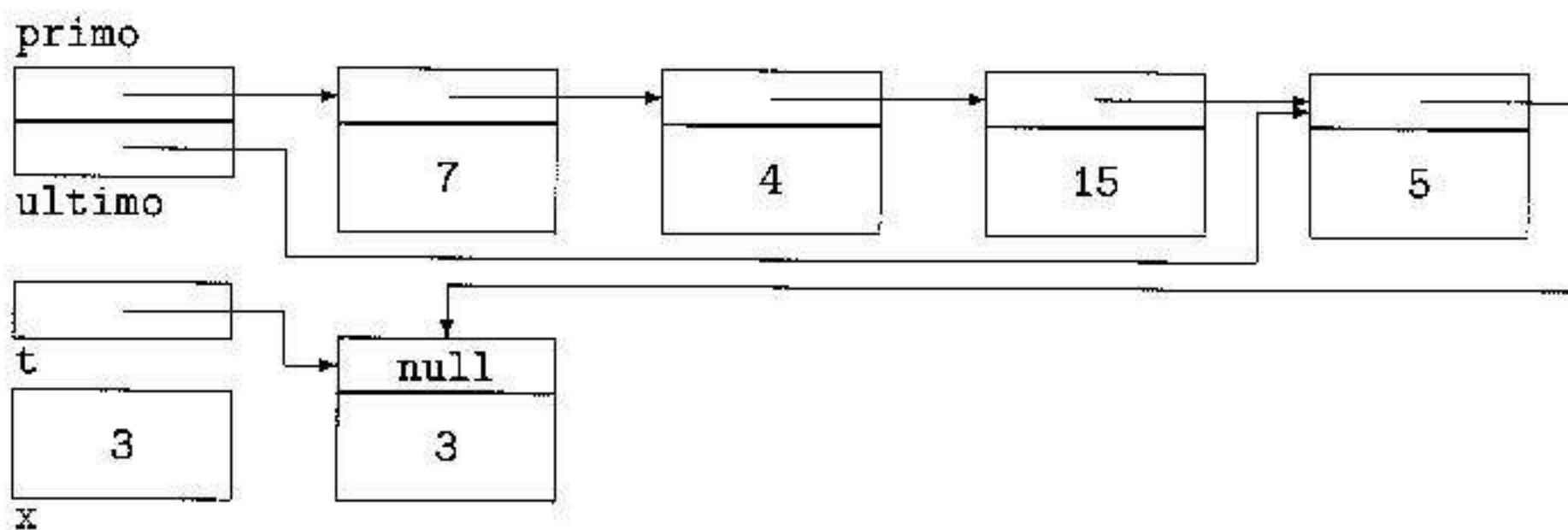
NodoCoda t = new NodoCoda();
t.dato = x;
t.pros = null;

```

Dopo l'esecuzione di queste operazioni la memoria contiene:

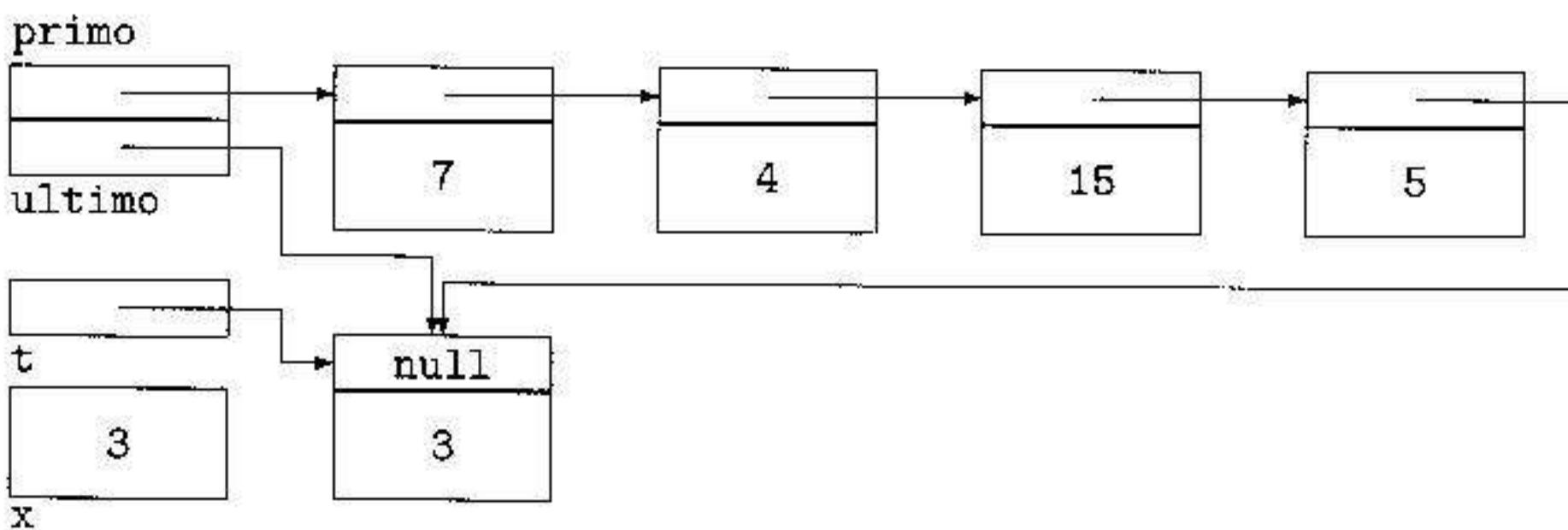


- Collegamento del nuovo nodo alla fine della lista (dopo il nodo puntato da **ultimo**) mediante l'assegnamento **ultimo.pros = t;**



A questo punto il nuovo nodo è stato correttamente inserito nella lista.

- Aggiornamento di **ultimo**, che deve anch'esso riferirsi al nuovo nodo, mediante l'assegnamento **ultimo = t**;



Se la coda è vuota, ambedue i campi, **primo** e **ultimo**, contengono **null**. In questo caso l'inserimento avviene facendo puntare sia **primo** sia **ultimo** al nuovo nodo creato, cioè scrivendo gli assegnamenti:

```
primo = ultimo = t;
```

Ecco il testo completo del metodo realizzato in questo modo:

```
public void aggiungi(E x) {
 //creazione del nuovo nodo
 NodoCoda t = new NodoCoda();
 t.dato = x;
 t.pros = null;
 //inserimento del nodo
 if (primo == null) //caso di coda inizialmente vuota
 primo = ultimo = t;
 else { //caso di coda non vuota
 ultimo.pros = t; //collega il nuovo nodo dopo l'ultimo
 }
}
```

```

 ultimo = t; //aggiorna il riferimento ultimo
}
}

```

Scriviamo ora un metodo `preleva` per prelevare il primo oggetto in coda e restituirne il riferimento. Nel caso la coda sia vuota, questo metodo non sarà in grado di svolgere il proprio compito. Gestiamo questa situazione sollevando un'eccezione. A tale scopo definiamo la seguente classe:

```

public class CodaVuotaException extends RuntimeException {
}

```

Lo schema del metodo `preleva` è dunque:

```

public E preleva() {
 if (la coda è vuota)
 throw new CodaVuotaException();
 else {
 elimina dalla coda il primo nodo
 return il riferimento contenuto nel campo dato del nodo eliminato
 }
}

```

Per controllare se la coda è vuota, è sufficiente eseguire il test `primo == null`. Se la coda non è vuota, l'eliminazione del primo elemento avviene in maniera analoga a quanto fatto per la pila:

- si memorizza il riferimento da restituire in una variabile `risultato` di tipo `E` mediante l'assegnamento `risultato = primo.dato`;
- si fa puntare il riferimento `primo` al secondo elemento scrivendo `primo = primo.pros`.

Che cosa succede se la coda, prima di eseguire `preleva`, contiene un solo nodo? Ambedue i campi, `primo` e `ultimo`, si riferiscono a tale nodo e il campo `primo.pros` contiene `null`. L'assegnamento `primo = primo.pros` assegna correttamente `null` al campo `primo`. Tuttavia il campo `ultimo` continua a riferirsi al nodo che è stato eliminato. In questo caso bisogna assegnare a `ultimo` il riferimento `null`, al fine di ottenere la coda vuota. Ecco il codice completo del metodo `preleva` costruito secondo questo schema:

```

public E preleva() {
 if (primo == null)
 throw new CodaVuotaException();
 else {
 E risultato = primo.dato;
 primo = primo.pros;
 if (primo == null) //caso in cui la coda sia rimasta vuota
 ultimo = null;
 }
}

```

```

 return risultato;
 }
}

```

Scriviamo ora un metodo `toString` che restituisca una stringa che rappresenti il contenuto della coda. La stringa dovrà contenere, separate da uno spazio, le rappresentazioni degli oggetti contenuti nella coda, dal primo all'ultimo. Ad esempio, per la coda della prima figura, la stringa dovrà essere:

7 4 15 5

Il metodo `toString` può scandire la coda dal primo all'ultimo elemento, costruendo man mano la stringa `s` da restituire. A tale stringa, inizialmente vuota, sarà concatenata a ogni passo una stringa che rappresenti il contenuto del nodo considerato, ottenibile utilizzando il metodo `toString` dell'oggetto memorizzato nel nodo, più lo spazio per la separazione dalla stringa successiva.

La scansione della coda può avvenire con l'uso di un ciclo `for` basato su un riferimento ausiliario di nome `nodo`. Tale riferimento è inizializzato copiandovi il puntatore `primo`. A ogni passo il riferimento viene spostato avanti, fino a raggiungere la fine della coda:

```

public String toString() {
 String s = "";
 for (NodoCoda nodo = primo; nodo != null; nodo = nodo.pros)
 s = s + nodo.dato.toString() + " ";
 return s;
}

```

È abbastanza utile permettere alle classi che usano Coda di scegliere come separare tra loro i contenuti dei vari nodi nella stringa costruita dal metodo `toString`. Ad esempio potrebbe essere comodo separarli con un trattino o con la stringa "\n" per ottenere di andare a capo. Scriviamo dunque un altro metodo `toString` che riceva, come parametro, la stringa da utilizzare per separare tra di loro le stringhe associate ai vari nodi. Modifichiamo anche la struttura del ciclo usato al fine di evitare di concatenare la stringa di separazione anche dopo l'ultimo nodo:

```

public String toString(String separatore) {
 if (primo == null)
 return "";
 else {
 String s = primo.dato.toString();
 for (NodoCoda nodo = primo.pros; nodo != null; nodo = nodo.pros)
 s = s + separatore + nodo.dato.toString();
 return s;
 }
}

```

Ora il metodo `toString` senza argomenti può essere facilmente riscritto in modo che si serva del metodo appena riportato.

Ecco il testo completo della classe `Coda`, nella quale è stato aggiunto anche un metodo `primo` che restituisce un riferimento al primo oggetto in coda, senza rimuoverlo, e un metodo vuota che permette di verificare se la coda è vuota:

```
public class Coda<E> {
 private NodoCoda primo, ultimo;

 private class NodoCoda {
 E dato;
 NodoCoda pros;
 }

 public Coda() {
 primo = ultimo = null;
 }

 public void aggiungi(E x) {
 //creazione del nuovo nodo
 NodoCoda t = new NodoCoda();
 t.dato = x;
 t.pros = null;
 //inserimento del nodo
 if (primo == null) //caso di coda inizialmente vuota
 primo = ultimo = t;
 else { //caso di coda non vuota
 ultimo.pros = t; //collega il nuovo nodo dopo l'ultimo
 ultimo = t; //aggiorna il riferimento ultimo
 }
 }

 public E preleva() {
 if (primo == null)
 throw new CodaVuotaException();
 else {
 E risultato = primo.dato;
 primo = primo.pros;
 if (primo == null) //caso in cui la coda sia rimasta vuota
 ultimo = null;
 return risultato;
 }
 }
}
```

```

public E primo() {
 if (primo == null)
 throw new CodaVuotaException();
 else
 return primo.dato;
}

public String toString() {
 return this.toString(" ");
}

public String toString(String separatore) {
 if (primo == null)
 return "";
 else {
 String s = primo.dato.toString();
 for (NodoCoda nodo = primo.pros; nodo != null; nodo = nodo.pros)
 s = s + separatore + nodo.dato.toString();
 return s;
 }
}

public boolean vuota() {
 return primo == null;
}
}

```

Riportiamo ora una classe che permette di provare i metodi di Coda. La classe gestisce una coda inizialmente vuota. In base a un menu l'utente può scegliere di effettuare diverse operazioni sulla coda:

```

import prog.io.*;

public class ProvaCoda {
 private static ConsoleInputManager in = new ConsoleInputManager();
 private static ConsoleOutputManager out = new ConsoleOutputManager();

 public static void main(String[] args) {
 int pos, scelta;
 String riga;
 Coda<String> coda = new Coda<String>();
 while ((scelta = menu()) != 0)

```

```
switch (scelta) {
 case 1:
 riga = in.readLine("Riga da inserire? ");
 coda.aggiungi(riga);
 break;
 case 2:
 try {
 riga = coda.preleva();
 out.println("È stata eliminata dalla coda la riga:" + riga);
 } catch (CodaVuotaException e) {
 out.println("Impossibile eliminare elementi da una coda vuota");
 }
 break;
 case 3:
 try {
 riga = coda.primo();
 out.println("Il primo elemento della coda è " + riga);
 } catch (CodaVuotaException e) {
 out.println("La coda è vuota");
 }
 break;
 case 4:
 if (coda.vuota())
 out.println("Coda vuota");
 else {
 out.println("La coda contiene: ");
 out.println(coda.toString("\n"));
 }
 break;
 case 0:
 break;
 default:
 out.println("scelta non valida");
 break;
}
}

private static int menu() {
 out.println("\nScelte disponibili:\n");
 out.println("1. Inserimento di un elemento");
 out.println("2. Cancellazione del primo elemento");
 out.println("3. Visualizzazione primo elemento");
```

```

 out.println("4. Visualizzazione della coda");
 out.println();
 out.println("0. Uscita");
 out.println();
 int scelta = in.readInt("Scelta? ");
 return scelta;
}
}

```

## Esercizi

12.3 Scrivete un'applicazione che permetta di gestire i dati relativi a una coda di figure geometriche (rettangoli, quadrati, cerchi). All'inizio dell'esecuzione la coda è vuota. A ogni passo l'applicazione propone all'utente un menu con una serie di possibilità.

- Inserimento di una nuova figura in fondo alla coda.
- Eliminazione della prima figura in coda.
- Visualizzazione delle caratteristiche della figura con area maggiore tra quelle presenti nella coda.
- Visualizzazione delle caratteristiche della figura con perimetro maggiore tra quelle presenti nella coda.
- Calcolo e visualizzazione della somma delle aree delle figure presenti nella coda.
- Calcolo e visualizzazione della somma dei perimetri delle figure presenti nella coda.
- Visualizzazione dell'elenco delle figure presenti nella coda.
- Uscita dal programma.

Per svolgere l'esercizio utilizzate le classi `Figura`, `Rettangolo`, `Quadrato` e `Cerchio`. Per rappresentare la coda di figure estendete la classe `Coda` definendo una nuova classe contenente i metodi aggiuntivi necessari.

12.4 Scrivete un'implementazione della classe `Sequenza`, basata su una struttura a coda.

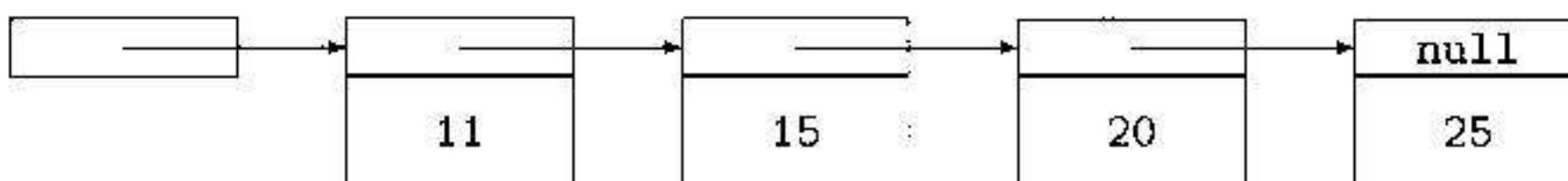
## 12.3 Le liste ordinate

Nelle strutture a pila e a coda la posizione degli elementi viene determinata in base al loro ordine di arrivo. In molte applicazioni, invece, la posizione degli elementi dev'essere determinata in base a una relazione di ordinamento: si pensi ad esempio a un elenco di nomi in ordine alfabetico o a un elenco di numeri in ordine crescente.

L'utilizzo di array per memorizzare sequenze ordinate di elementi è poco efficiente, perché l'inserimento di un nuovo elemento richiede lo spostamento di tutti gli elementi che lo seguono.

Al contrario, l'uso di strutture a lista, simili a quelle utilizzate per implementare pile e code, permette di realizzare efficientemente le operazioni di inserimento.

Studieremo ora le *liste ordinate* in cui le operazioni di inserimento vengono effettuate in modo che la struttura sia mantenuta ordinata. La seguente figura rappresenta una lista ordinata contenente gli interi 11 15 20 25:



Osserviamo anzitutto che per poter mantenere ordinata la struttura deve esistere tra gli elementi della struttura stessa una relazione d'ordine (ad esempio tra le stringhe esiste l'ordine alfabetico). Non tutti i tipi di oggetti prevedono una relazione d'ordine; pertanto il tipo E dei dati contenuti nelle liste ordinate non potrà essere un tipo qualunque. D'altra parte abbiamo visto che ogni classe che implementa l'interfaccia Comparable ammette una relazione d'ordine (gli oggetti della classe dispongono di un metodo compareTo per effettuare i confronti). Pertanto costruiremo una classe ListaOrdinata, nella quale i dati memorizzati saranno istanze di classi di un tipo parametro E che implementi l'interfaccia Comparable. Più precisamente, come discusso nel Paragrafo 6.17, richiederemo che il tipo parametro E estenda Comparable<? super E>.

Come per la classe Stack, definiamo una classe interna per rappresentare i nodi della lista. Inoltre definiamo un campo inizio destinato a contenere il riferimento al primo elemento della lista. Il costruttore della classe crea una lista vuota assegnando a inizio il riferimento null:

```

public class ListaOrdinata<E extends Comparable<? super E>> {
 private NodoLista inizio;

 private class NodoLista {
 E dato;
 NodoLista pros;
 }

 public ListaOrdinata() {
 inizio = null;
 }

 ...metodi...
}

```

Prima di tutto costruiamo un metodo int trova(E x) che cerchi nella lista un oggetto fornito tramite il parametro e ne restituisca la posizione. Cercando ad esempio 20 nella lista precedente dovrà essere restituito come risultato 3. Nel caso la lista non contenga l'elemento cercato, il metodo dovrà restituire 0.

Il metodo usa un riferimento ausiliario p per scandire la lista. All'inizio viene copiato in p il campo inizio, in modo che p si riferisca al primo nodo della lista. Successivamente viene

scandita la lista contando il numero di nodi traversati alla ricerca del nodo desiderato. Poiché la lista è ordinata, si può terminare la scansione quando si raggiunge la fine della lista o un nodo contenente un valore maggiore o uguale a quello cercato:

```
p = inizio;
posizione = 1;
while (la lista non è finita e il nodo puntato da p contiene
 un oggetto minore di quello cercato) {
 sposta p in avanti
 incrementa posizione
}
```

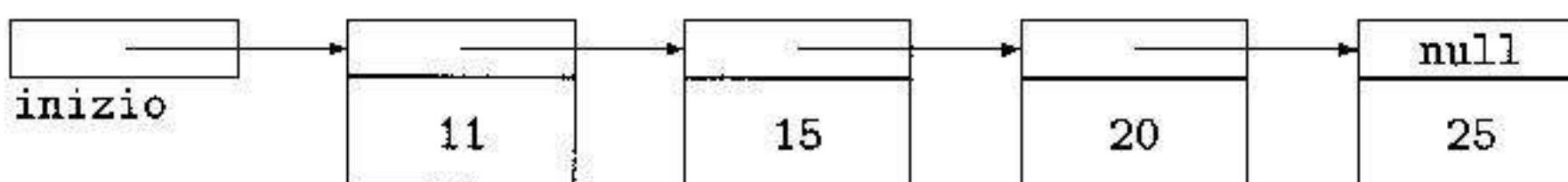
Al termine della scansione si verifica una delle seguenti situazioni:

- si è raggiunta la fine della lista (`p == null`); questo significa che l'oggetto non c'è, e quindi si deve restituire 0;
- si è raggiunto un oggetto diverso da quello cercato (in particolare maggiore di quello cercato); anche in questo caso si deve restituire 0;
- si è raggiunto l'oggetto cercato; in questo caso si restituisce il valore di `posizione`.

Ecco il codice completo del metodo:

```
public int trova(E x) {
 NodoLista p = inizio;
 int posizione = 1;
 while (p != null && p.dato.compareTo(x) < 0) {
 p = p.pros;
 posizione++;
 }
 if (p == null || p.dato.compareTo(x) > 0) //se non c'è
 return 0;
 else
 return posizione;
}
```

Costruiamo ora un metodo `void inserisci(E x)` per l'inserimento in una lista ordinata di un nuovo oggetto fornito tramite il parametro. Il metodo dovrà inserire l'oggetto nella posizione opportuna in base all'ordinamento. Ad esempio, dovendo inserire l'intero 18 nella seguente lista, occorrerà inserire un nuovo nodo tra il nodo contenente 15 e il nodo contenente 20.



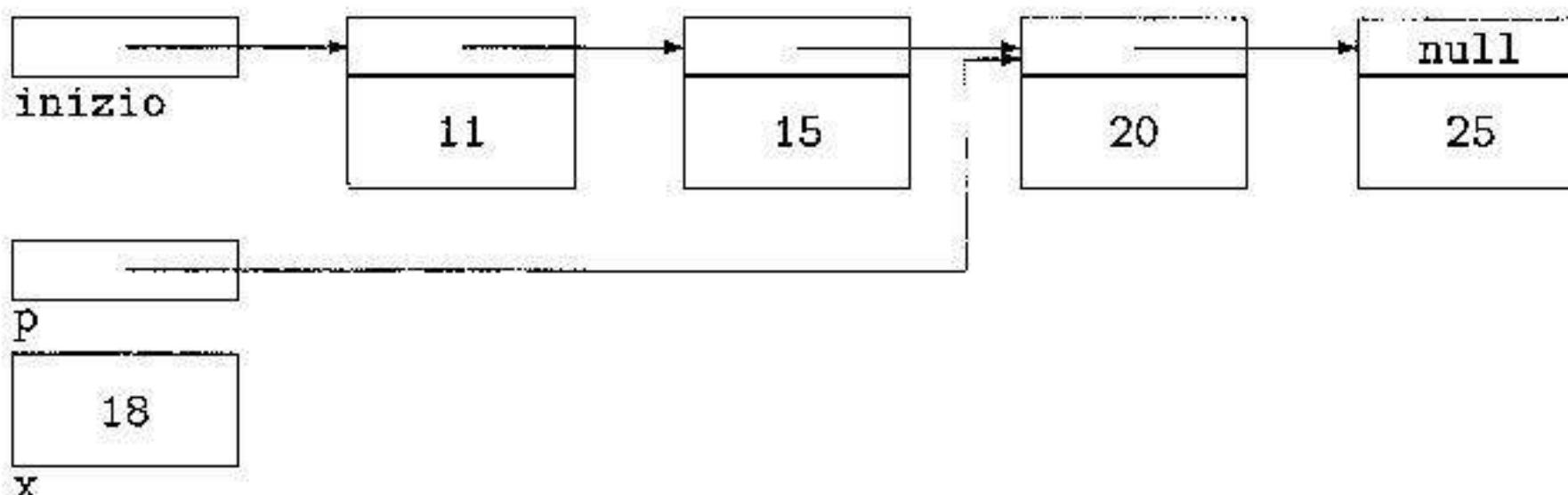
Il metodo è composto di due parti fondamentali:

- (1) ricerca della posizione dove effettuare l'inserimento;
- (2) creazione e inserimento del nuovo nodo.

La fase di ricerca è analoga a quella utilizzata nel metodo `trova`; in questo caso non è però necessario il contatore `posizione`:

```
p = inizio;
while (p != null && p.dato.compareTo(x) < 0)
 p = p.pros;
```

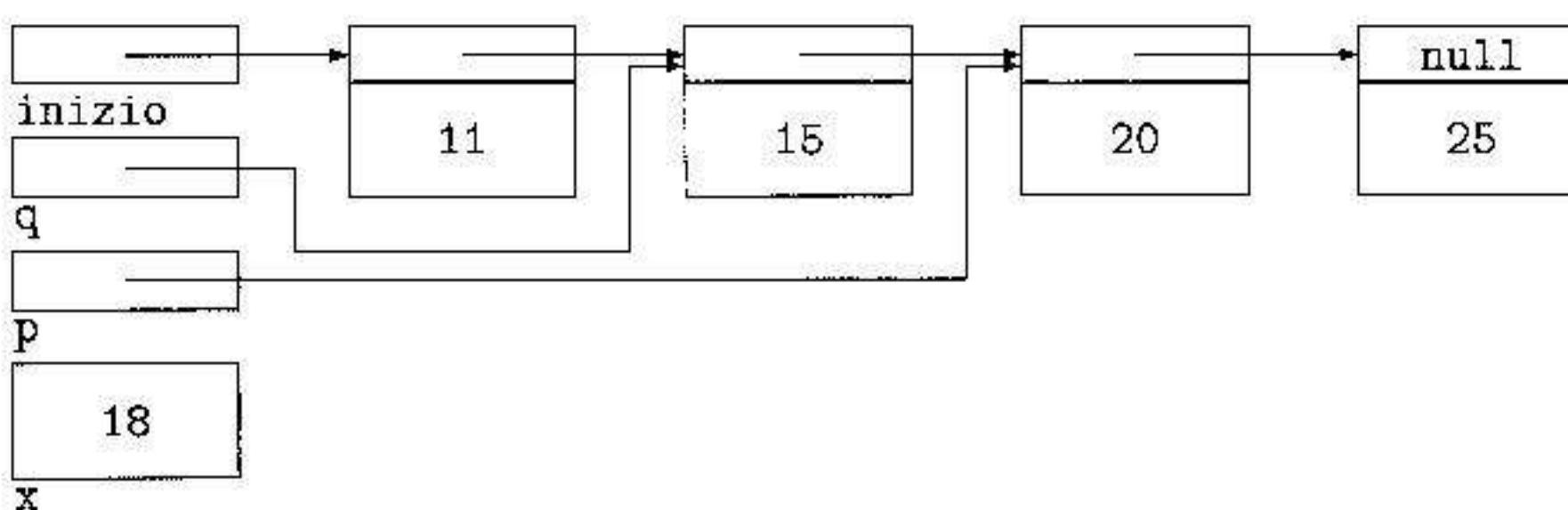
All'uscita da questo ciclo la variabile `p` contiene un riferimento al nodo *prima* del quale effettuare l'inserimento (o `null` nel caso l'inserimento vada fatto alla fine della lista).



Per effettuare l'inserimento occorre modificare il campo `pros` del nodo che *precede* il nodo riferito da `p`. Per disporre di un riferimento a tale nodo introduciamo un secondo riferimento ausiliario `q`, inizializzato a `null`, che a ogni passo della fase di ricerca farà riferimento al nodo che precede il nodo riferito da `p`. Introducendo `q`, la ricerca viene riscritta come:

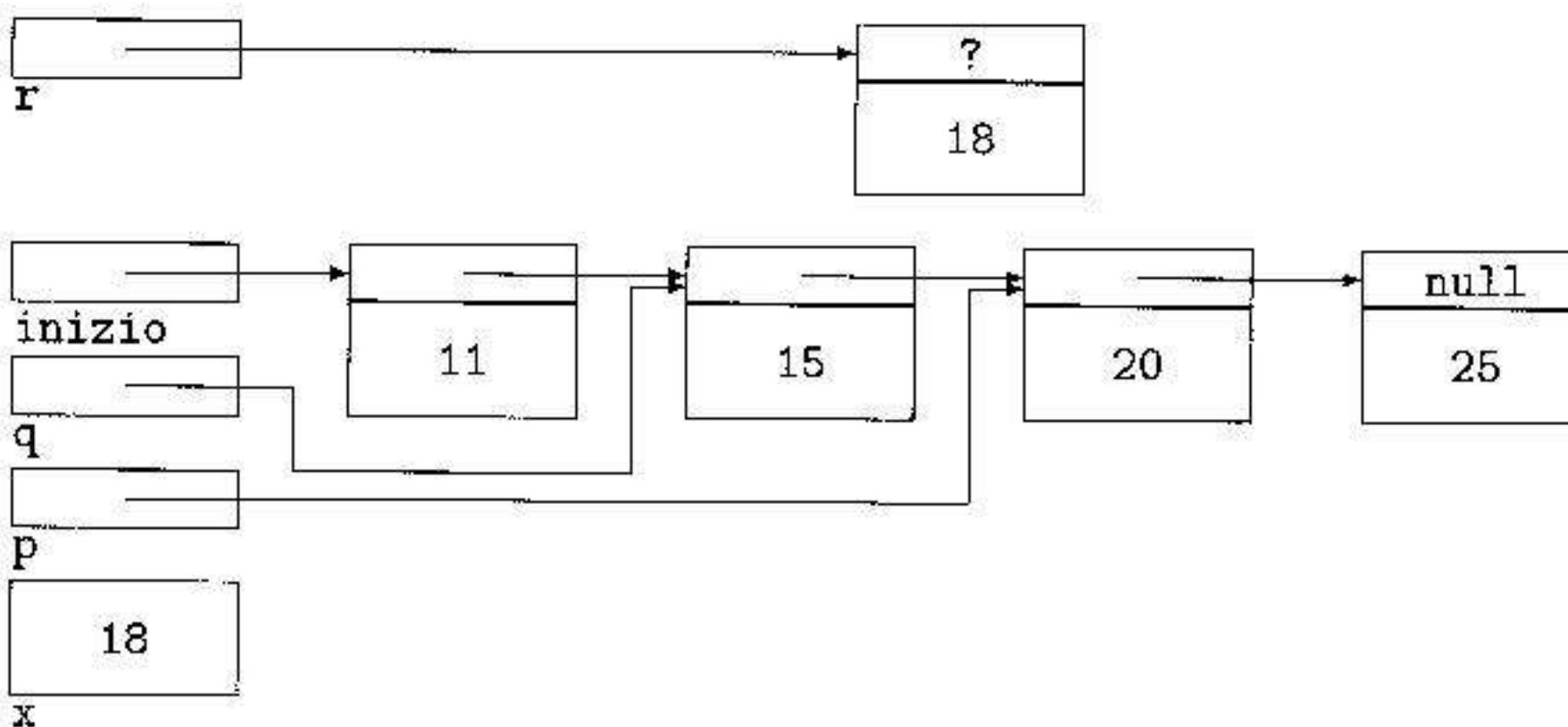
```
NodoLista p = inizio, q = null;
while (p != null && p.dato.compareTo(x) < 0) {
 q = p;
 p = p.pros;
}
```

Dopo l'esecuzione del ciclo si otterrà in questo caso:



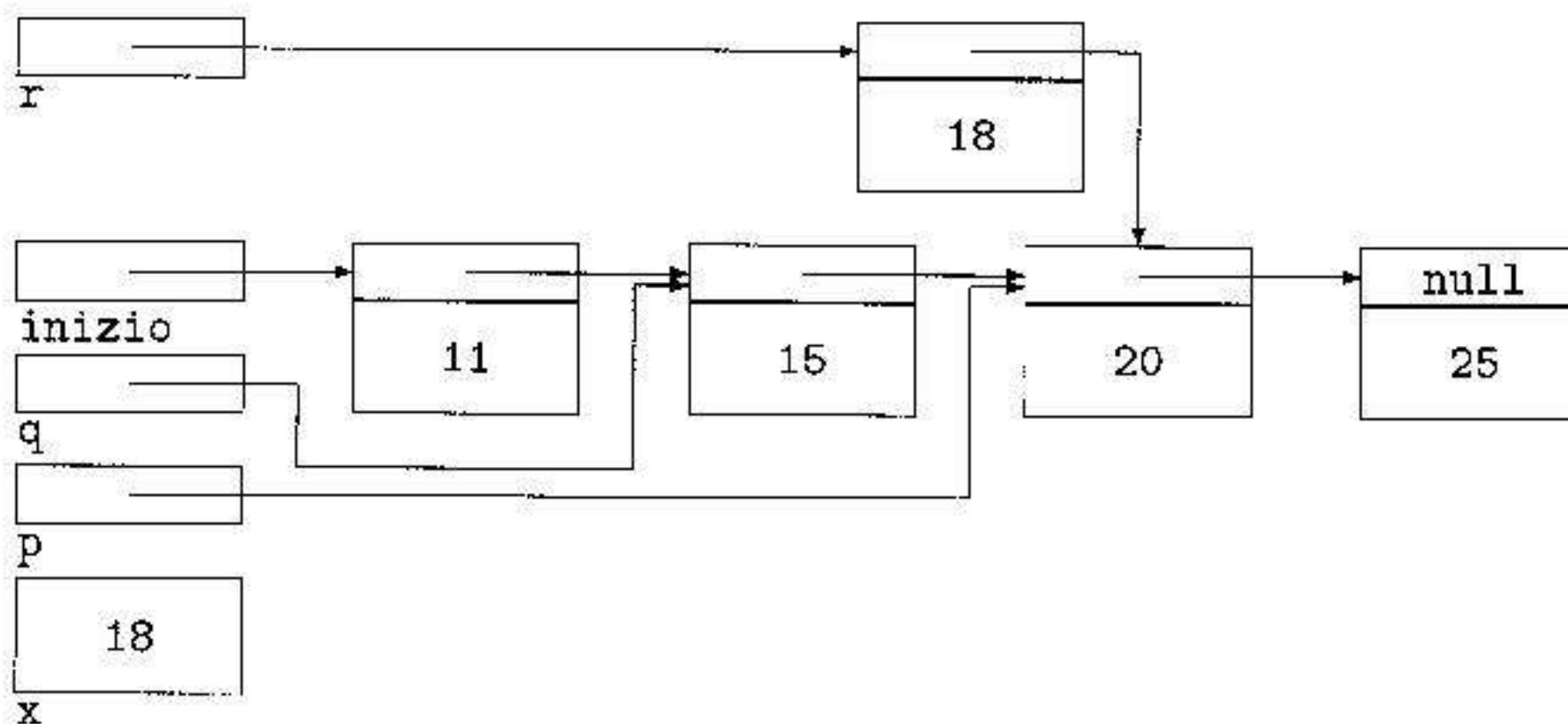
Terminata la fase di ricerca, occorre effettuare la creazione e l'inscimento del nodo. Per la creazione del nodo utilizziamo un terzo riferimento ausiliario **r** e scriviamo:

```
NodoLista r = new NodoLista();
r.dato = x;
```

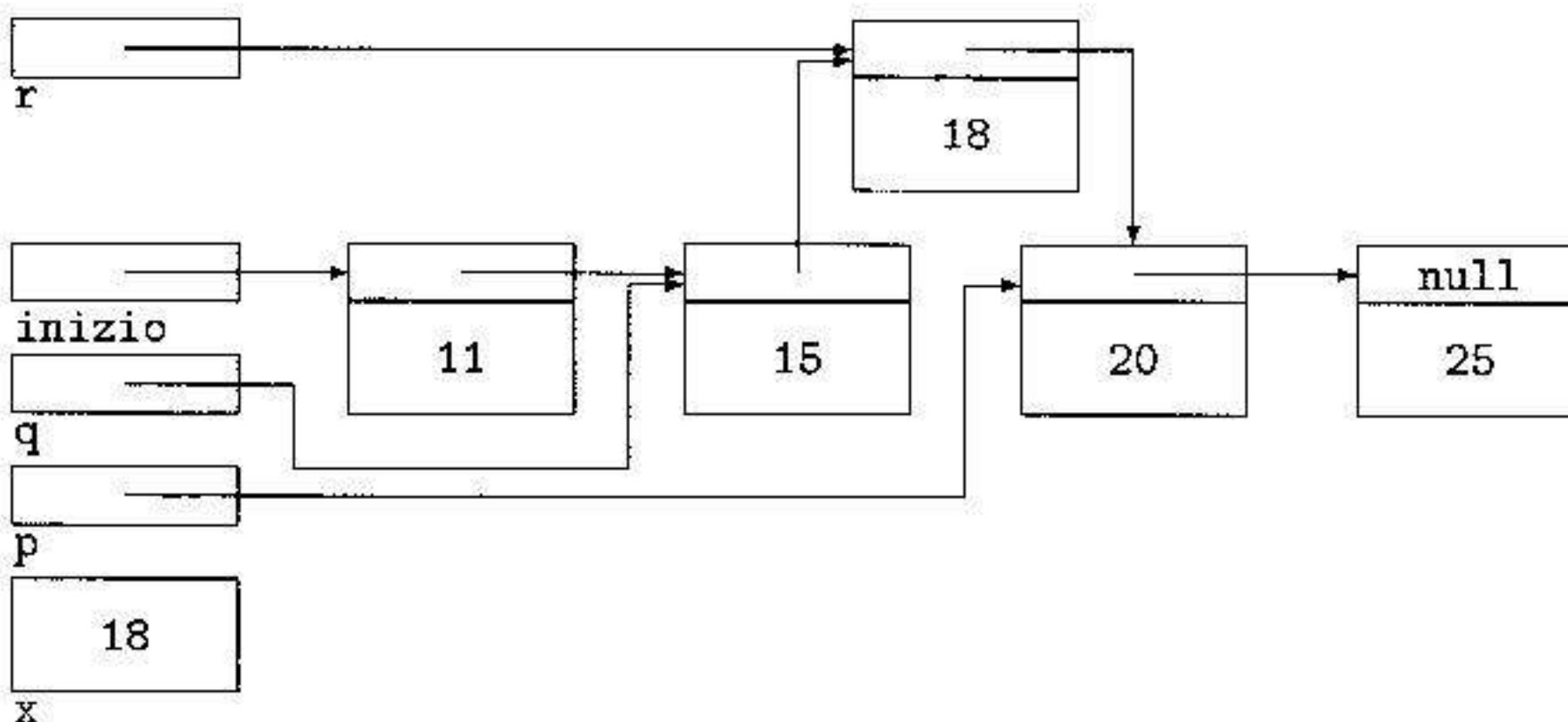


Per inserire il nuovo nodo *prima* del nodo puntato da **p** e *dopo* il nodo puntato da **q** occorre effettuare queste operazioni.

- (1) Agganciare la parte di lista riferita da **p** dopo il nuovo nodo (**r.pros** = **p**).



- (2) Agganciare il nuovo nodo dopo il nodo riferito da **q** (**q.pros** = **r**).



Se l'inserimento avviene immediatamente all'inizio della lista (dunque se **q == null**), occorre invece fare in modo che il puntatore iniziale della lista, cioè il campo **inizio**, si riferisca al nuovo nodo (**inizio = r**).

La seconda fase del metodo può quindi essere scritta come:

```
//creazione del nuovo nodo
NodoLista r = new NodoLista();
r.dato = x;

//inserimento del nodo nella lista tra i nodi riferiti da q e da p
r.pros = p;
if (q == null) //inserimento all'inizio
 inizio = r;
else //inserimento dopo il nodo riferito da q
 q.pros = r;
```

Il metodo completo è:

```
public void inserisci(E x) {
 //ricerca della posizione per l'inserimento
 NodoLista p = inizio, q = null;
 while (p != null && p.dato.compareTo(x) < 0) {
 q = p;
 p = p.pros;
 }

 //creazione del nuovo nodo
 NodoLista r = new NodoLista();
 r.dato = x;
```

```

//inserimento del nodo nella lista tra i nodi riferiti da q e da p
r.pros = p;
if (q == null) //inserimento all'inizio
 inizio = r;
else //inserimento dopo il nodo riferito da q
 q.pros = r;
}

```

Descriviamo ora la *cancellazione* di un elemento da una lista ordinata. Vogliamo costruire un metodo `void cancella(E x)` che riceva come parametro un riferimento `x` ed elimini dalla lista ordinata la prima occorrenza di un oggetto uguale a quello riferito da `x`, se presente. Ad esempio, data la lista contenente i valori 2 4 11 11 20, cancellando 11 il contenuto della lista diventerà 2 4 11 20.

La struttura del metodo è simile a quella del metodo di inserimento:

- (1) ricerca della posizione dove effettuare la cancellazione;
- (2) rimozione dell'elemento (da effettuare solo nel caso l'elemento sia stato trovato).

Ad alto livello lo schema del metodo sarà dunque:

```

cerca l'elemento da cancellare e determinane la posizione
if (l'elemento è stato trovato)
 eliminalo

```

La fase di ricerca è identica a quella del metodo `inserisci` e utilizza due riferimenti ausiliari, `p` e `q`:

```

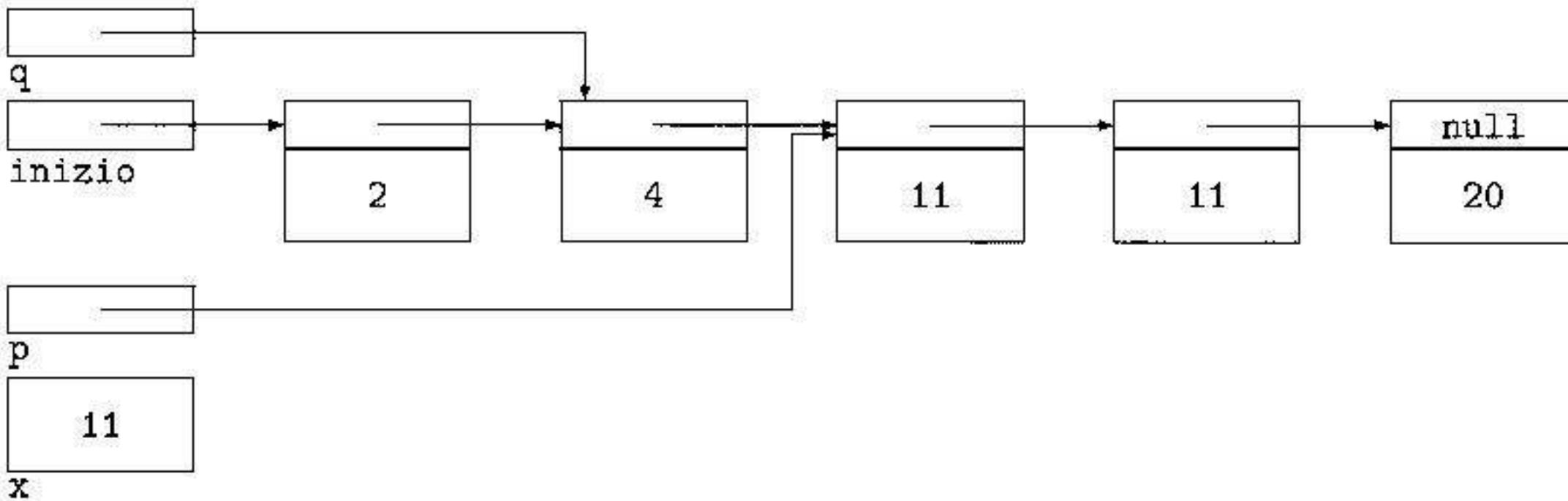
NodoLista p = inizio, q = null;
while (p != null && p.dato.compareTo(x) < 0) {
 q = p;
 p = p.pros;
}

```

Al termine dell'esecuzione del ciclo possono verificarsi le seguenti situazioni:

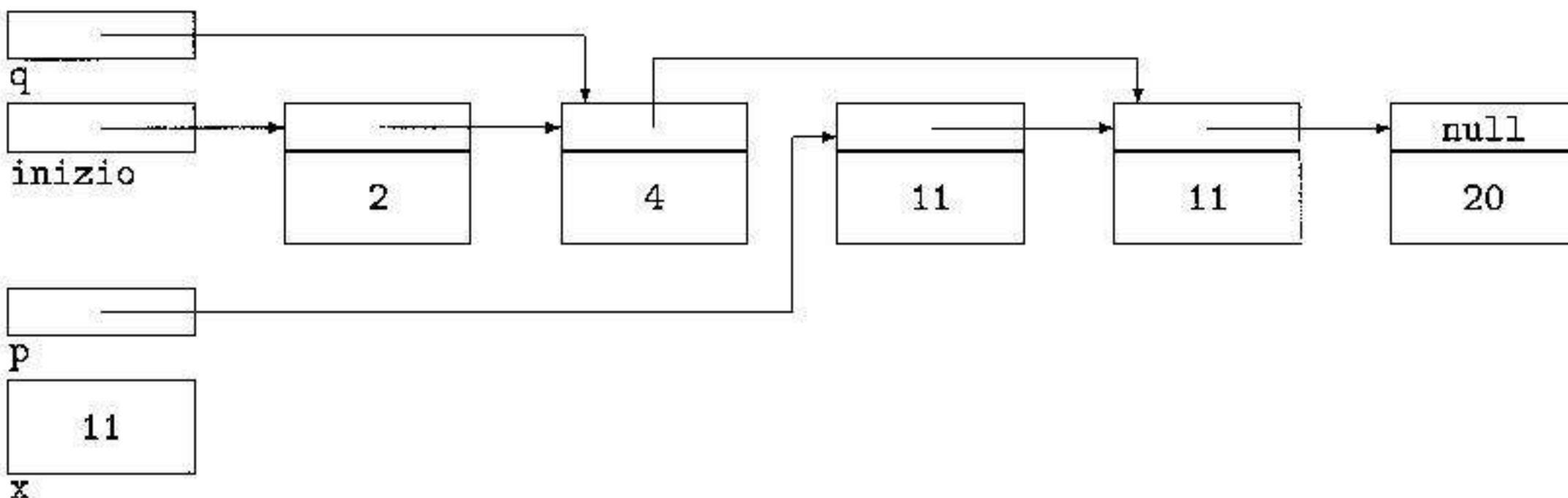
- il riferimento `p` contiene `null`, e dunque l'elemento non è stato trovato; in questo caso non sono necessarie altre operazioni;
- il riferimento `p` contiene il riferimento a un oggetto diverso (e dunque maggiore) da quello cercato; questo significa che l'oggetto non è presente nella lista, e quindi non occorre procedere alla cancellazione;
- il nodo riferito da `p` contiene il riferimento a un oggetto uguale a quello da eliminare; in questo caso bisogna procedere alla cancellazione del nodo.

Analizziamo in dettaglio l'ultimo caso.



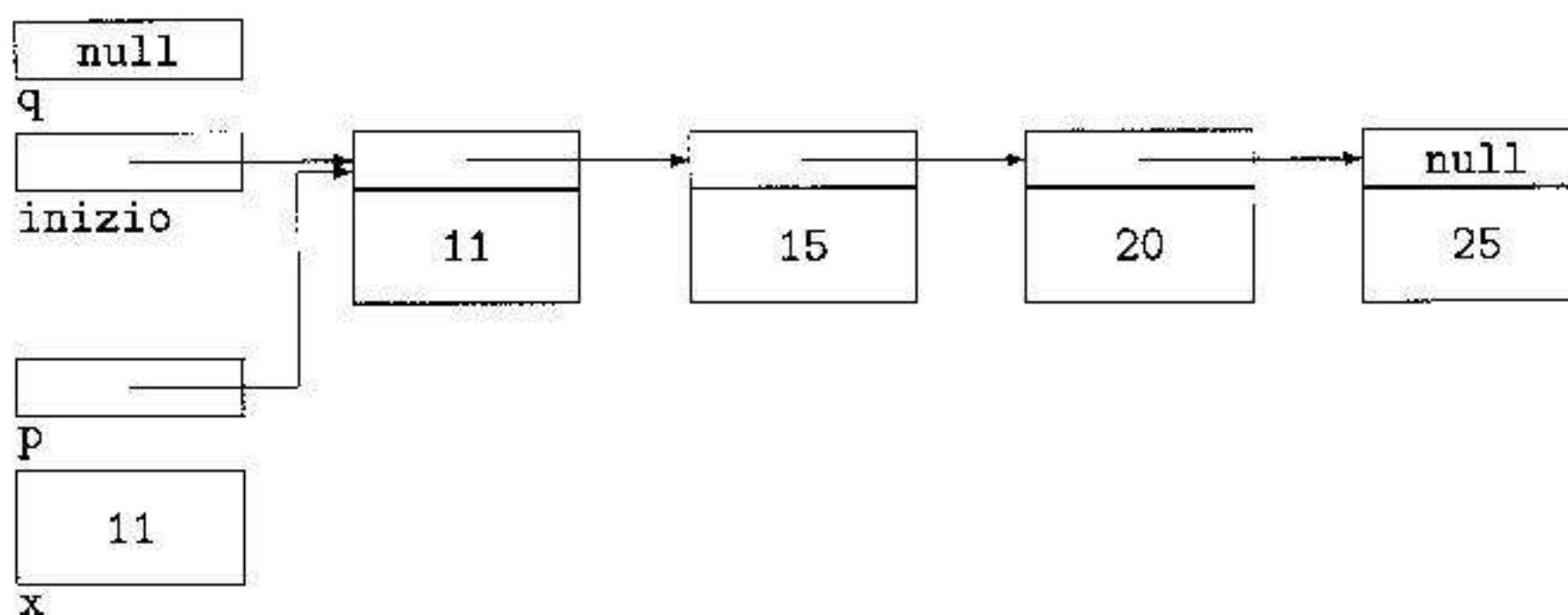
Per sganciare dalla lista il nodo riferito da `p` è necessario collegare il nodo che lo precede (che è riferito da `q`) a quello che lo segue. A tale scopo basta scrivere l'assegnamento:

```
q.pros = p.pros
```

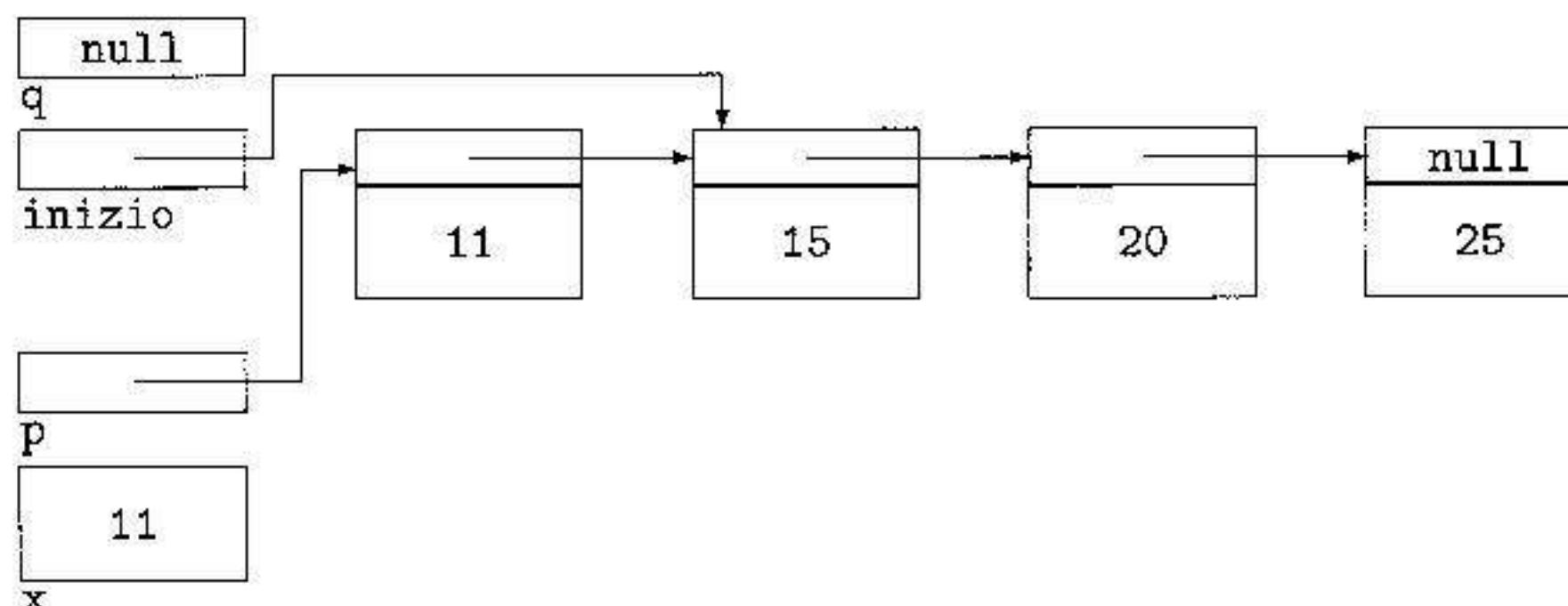


Analizziamo ora due casi particolari.

- Se il nodo da eliminare è l'ultimo della lista, cioè se `p.pros` contiene `null`, l'effetto dell'assegnamento `q.pros = p.pros` è di assegnare `null` a `q.pros`, facendo dunque terminare la lista sul nodo riferito da `q`.
- Se il nodo da eliminare è il primo della lista, dopo la ricerca `q` contiene `null`, ed entrambi, `p` e `inizio`, si riferiscono al primo elemento della lista. Ad esempio, volendo cancellare 11 dalla lista contenente 11 15 20 25, dopo la ricerca i riferimenti saranno organizzati come nella seguente figura:



In questo caso, per sganciare il nodo dalla lista basta far puntare `inizio` al secondo nodo mediante l'assegnamento `inizio = inizio.pros`:



Riassumendo: la fase di cancellazione viene effettuata se la lista contiene un nodo con il valore cercato, cioè se, al termine dell'esecuzione del ciclo, il riferimento `p` è diverso da `null` e l'oggetto al quale fa riferimento `p.dato` è uguale a quello cui fa riferimento `x`. In questo caso si possono verificare due situazioni differenti:

- il nodo da eliminare è il primo: si modifica direttamente il puntatore `inizio`;
- il nodo da eliminare è successivo al primo: si modifica il campo `pros` del nodo che precede quello da eliminare.

```

if (p != null && p.dato.equals(x))
 if (q == null) //cancellazione all'inizio
 inizio = inizio.pros;
 else //cancellazione dopo il nodo riferito da q
 q.pros = p.pros;
 }
}

```

Il testo completo del metodo è dunque:

```
public void cancella(E x) {
 //ricerca della posizione per l'inserimento
 NodoLista p = inizio, q = null;
 while (p != null && p.dato.compareTo(x) < 0) {
 q = p;
 p = p.pros;
 }

 //eliminazione del nodo
 if (p != null && p.dato.equals(x))
 if (q == null) //cancellazione all'inizio
 inizio = inizio.pros;
 else //cancellazione dopo il nodo riferito da q
 q.pros = p.pros;
}
```

Ecco il testo completo della classe ListaOrdinata:

```
public class ListaOrdinata<E extends Comparable<? super E>> {
 private NodoLista inizio;

 private class NodoLista {
 E dato;
 NodoLista pros;
 }

 /* costruisce una lista vuota */
 public ListaOrdinata() {
 inizio = null;
 }

 /* restituisce la posizione di un oggetto nella lista
 o 0, nel caso l'oggetto non sia presente */
 public int trova(E x) {
 NodoLista p = inizio;
 int posizione = 1;
 while (p != null && p.dato.compareTo(x) < 0) {
 p = p.pros;
 posizione++;
 }
 if (p == null || p.dato.compareTo(x) > 0) //se non c'è
 return 0;
 }
}
```

```
 else
 return posizione;
}

/* inserisce un nuovo elemento nella lista */
public void inserisci(E x) {
 //ricerca della posizione per l'inserimento
 NodoLista p = inizio, q = null;
 while (p != null && p.dato.compareTo(x) < 0) {
 q = p;
 p = p.pros;
 }

 //creazione del nuovo nodo
 NodoLista r = new NodoLista();
 r.dato = x;

 //inserimento del nodo nella lista tra i nodi riferiti da q e da p
 r.pros = p;
 if (q == null) //inserimento all'inizio
 inizio = r;
 else //inserimento dopo il nodo riferito da q
 q.pros = r;
}

/* cancella un elemento dalla lista */
public void cancella(E x) {
 //ricerca della posizione per l'inserimento
 NodoLista p = inizio, q = null;
 while (p != null && p.dato.compareTo(x) < 0) {
 q = p;
 p = p.pros;
 }

 //eliminazione del nodo
 if (p != null && p.dato.equals(x))
 if (q == null) //cancellazione all'inizio
 inizio = inizio.pros;
 else //cancellazione dopo il nodo riferito da q
 q.pros = p.pros;
}
```

```
/* restituisce una stringa corrispondente al contenuto della lista */
public String toString() {
 return this.toString(" ");
}

/* restituisce una stringa corrispondente al contenuto della lista;
 utilizzando l'argomento per separare i vari elementi */
public String toString(String separatore) {
 if (inizio == null)
 return "";
 else {
 String s = inizio.dato.toString();
 for (NodoLista nodo = inizio.pros; nodo != null; nodo = nodo.pros)
 s = s + separatore + nodo.dato.toString();
 return s;
 }
}

/* restituisce true se la lista è vuota */
public boolean vuota() {
 return inizio == null;
}
```

Ecco il testo di un'applicazione di prova per i metodi di ListaOrdinata:

```
import prog.io.*;

public class ProvaListaOrdinata {
 private static ConsoleInputManager in = new ConsoleInputManager();
 private static ConsoleOutputManager out = new ConsoleOutputManager();

 public static void main(String[] args) {
 String stringa;
 int pos, scelta;
 ListaOrdinata<String> lista = new ListaOrdinata<String>();
 while ((scelta = menu()) != 0)
 switch (scelta) {
 case 1:
 stringa = in.readLine("Stringa da inserire? ");
 lista.inserisci(stringa);
 break;
 case 2:
```

```
stringa = in.readLine("Stringa da cancellare? ");
lista.cancella(stringa);
break;
case 3:
 stringa = in.readLine("Stringa da cercare? ");
 pos = lista.trova(stringa);
 if (pos == 0)
 out.println("La stringa \" "+ stringa +
 "\" non è presente");
 else
 out.println("La stringa \" "+ stringa +
 "\" si trova in posizione " + pos);
 break;
case 4:
 if (lista.vuota())
 out.println("Non vi è alcuna stringa memorizzata");
 else {
 out.println("Elenco stringhe memorizzate:");
 out.println(lista.toString("\n"));
 }
 break;
case 0:
 break;
default:
 out.println("scelta non valida");
 break;
}
}

private static int menu() {
 out.println("\nScelte disponibili:\n");
 out.println("1. Inserimento di una stringa");
 out.println("2. Cancellazione di una stringa");
 out.println("3. Ricerca di una stringa");
 out.println("4. Visualizzazione delle stringhe memorizzate");
 out.println();
 out.println("0. Uscita");
 out.println();
 int scelta = in.readInt("Scelta? ");
 return scelta;
}
```

## Esercizi

12.5 Costruite una classe `ListaDisordinata` che realizzi una lista di oggetti in cui gli elementi sono sempre inseriti all'inizio della lista. Oltre a un metodo per l'inserimento di nuovi elementi e a metodi analoghi a quelli di `ListaOrdinata`, scrivete un metodo che trasformi la lista facendo scorrere circolarmente all'indietro tutti gli elementi: tutti gli elementi vengono spostati indietro di una posizione, eccetto il primo, che passa nell'ultima. Ad esempio la lista contenente 54 23 654 12 26 dovrà essere trasformata nella lista contenente 23 654 12 26 54.

12.6 Costruite una classe `ListaInteri` che realizzi una lista di interi in cui gli elementi vengono inseriti sempre *all'inizio* della lista. Oltre a un metodo per l'inserimento di nuovi elementi e a metodi analoghi a quelli di `ListaOrdinata`, implementate i seguenti metodi.

- Un metodo per scambiare l'elemento minimo della lista con il massimo. Ad esempio la lista contenente 24 3 7 98 46 22 dovrà essere trasformata nella lista contenente 24 98 7 3 46 22.
- Un metodo che raddoppi i valori contenuti nei nodi. Ad esempio, la lista contenente 5 8 10 25 dovrà essere trasformata nella lista contenente 10 16 20 50.
- Un metodo che restituisca la somma dei valori presenti nella lista.
- Un metodo che restituisca la media dei valori presenti nella lista, e sollevi un'eccezione nel caso di lista vuota.

Scrivete poi un'applicazione di prova per questi metodi.

12.7 Si considerino le seguenti classi (le istanze della prima classe, di cui sono evidenziate solo alcune parti, rappresentano liste di interi):

```
public class ListaInteri {
 private NodoLista inizio;

 private class NodoLista {
 int dato;
 NodoLista pros;
 }

 public ListaInteri() {
 inizio = null;
 }
 ...metodi...
}
```

```
public class ListaInteriException
 extends RuntimeException {

 public ListaInteriException(String s) {
 super(s);
 }
}
```

Scrivete un metodo **maxPari** (che sarà collocato nella classe **ListaInteri**), il cui compito è quello di restituire il più grande valore pari presente nella lista. Se la lista è vuota o non contiene numeri pari, il metodo deve sollevare una **ListaInteriException**.

Scrivete un metodo **visualizzaMaxPari**, da collocare sempre nella classe **ListaInteri**, che visualizzi il massimo valore pari presente nella lista. Per determinare tale valore il metodo deve richiamare **maxPari**. Il metodo deve inoltre essere in grado di intercettare l'eccezione che può essere sollevata da **maxPari** visualizzando in questo caso un messaggio d'errore.

- 12.8 Scrivete una classe **Stringa** contenente alcuni metodi di **String**, come ad esempio **concat**, **length** e **substring**. Per rappresentare una stringa utilizzate una lista di caratteri.
- 12.9 Il metodo **trova** della classe **ListaOrdinata** restituisce la posizione di un oggetto nella lista. Ampliate la classe scrivendo un metodo **trovaOggetto** che riceva un parametro di tipo **E** e restituisca il riferimento a un oggetto memorizzato nella lista uguale a quello fornito tramite il parametro. Se l'oggetto non è presente, il metodo deve restituire **null**.
- 12.10 Il metodo **inserisci** della classe **ListaOrdinata** inserisce un elemento anche se è già presente nella lista. Ampliate la classe **ListaOrdinata** scrivendo un metodo **inserisciSeManca** che effettui l'inserimento solo se l'elemento non è già presente.
- 12.11 Il metodo **cancella** della classe **ListaOrdinata** cancella solo la prima occorrenza di un valore. Ampliate la classe **ListaOrdinata** scrivendo un metodo **cancellaTutti** che cancelli da una lista ordinata tutte le occorrenze dell'elemento che si desidera cancellare.
- 12.12 Ampliate la classe **ListaOrdinata** scrivendo un metodo **eliminaRipetizioni** che modifichi la lista eliminando tutte le ripetizioni di elementi. Ad esempio, se la lista contiene inizialmente 1 5 5 8 8 8 10 14 14, dopo l'esecuzione del metodo dovrà contenere 1 5 8 10 14.
- 12.13 Scrivete un'applicazione che permetta di provare l'esecuzione dei metodi di **ListaOrdinata** e dei nuovi metodi introdotti negli esercizi precedenti.

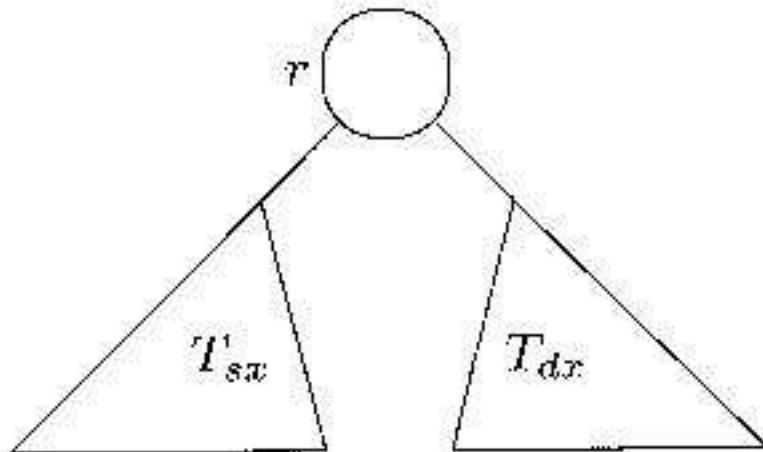
## 12.4 Alberi binari

Altre strutture dati dinamiche fondamentali, insieme alle liste, sono gli *alberi binari*. La definizione di albero binario viene data in maniera ricorsiva come segue:

**Definizione** Un albero binario è:

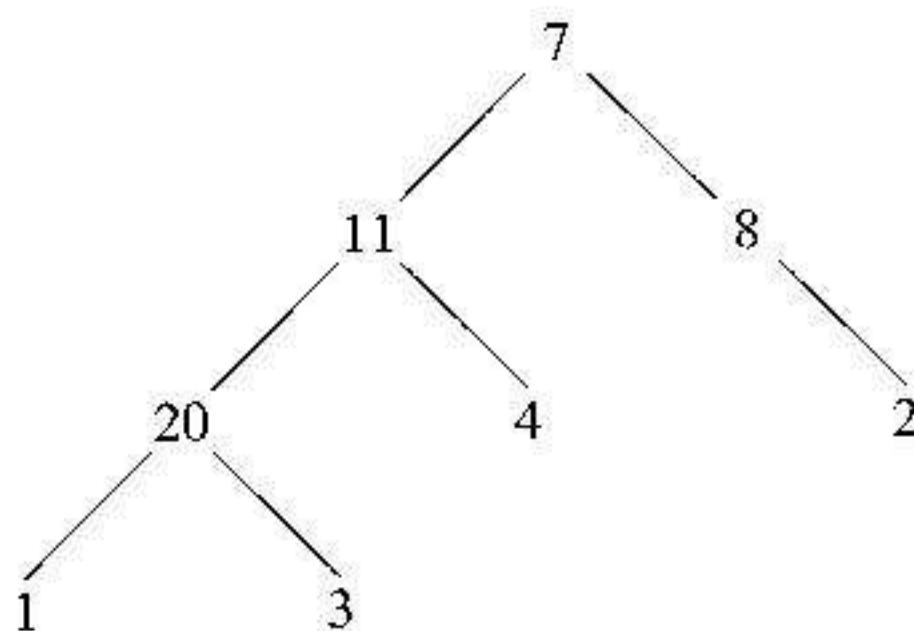
- la struttura vuota, oppure
- un nodo, detto radice, cui sono associate due strutture ad albero denominate, rispettivamente, sottoalbero sinistro e sottoalbero destro.

In base alla definizione, un albero binario non vuoto  $T$  ha la struttura rappresentata nella seguente figura, dove  $r$  è il nodo *radice* e  $T_{sx}$  e  $T_{dx}$  sono, rispettivamente, il *sottoalbero sinistro* e il *sottoalbero destro*:



Se gli alberi  $T_{sx}$  e  $T_{dx}$  sono a loro volta non vuoti, le loro radici sono dette *figli* del nodo  $r$ .

Questa figura mostra un albero contenente numeri interi:



In questo albero i figli del nodo contenente il valore 11 sono i nodi contenenti 20 e 4. Il nodo contenente 11 è anche detto *padre* di questi due nodi.

Una *foglia* è un nodo privo di figli. Nell'albero della figura, le foglie sono i nodi contenenti i valori 1, 2, 3 e 4.

Il *livello* di un nodo di un albero  $T$  è definito ricorsivamente come segue:

- il livello della *radice* di  $T$  è 1
- il livello dei figli di un nodo di livello  $i$  è  $i + 1$ .

La *profondità* di un albero è il massimo livello dei suoi nodi.

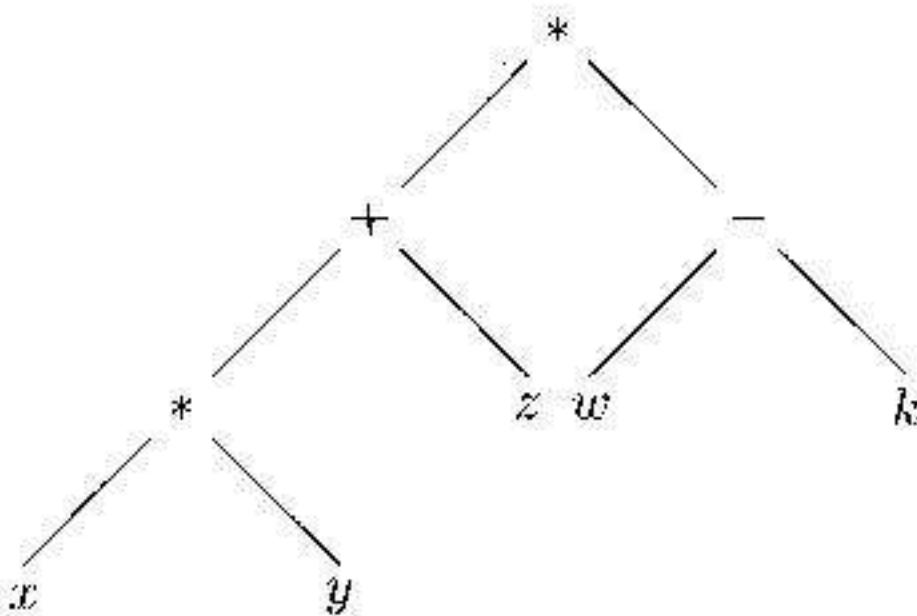
Nell'albero rappresentato nella figura precedente, il livello del nodo contenente 20 è 3, la profondità dell'albero 4. I nodi di livello 2 sono i nodi contenenti i valori 8 e 11.

Le strutture ad albero hanno svariate applicazioni. Con una struttura ad albero è possibile rappresentare organizzazioni gerarchiche, come ad esempio l'indice di un libro. Un libro infatti è diviso in capitoli, ognuno dei quali è diviso in paragrafi, che a loro volta possono essere suddivisi in sottoparagrafi. Possiamo rappresentare ognuna di queste entità con un nodo. La radice dell'albero rappresenta il Capitolo 1. Dato un nodo, rappresentiamolo alla sua destra i nodi successivi dello stesso livello, e alla sua sinistra i nodi di livello inferiore. Pertanto il figlio destro della radice corrisponderà al Capitolo 2, il figlio sinistro della radice al Paragrafo 1.1, il figlio destro del figlio sinistro della radice al Paragrafo 1.2, e così via.

Le espressioni aritmetiche hanno una rappresentazione naturale sotto forma di alberi binari. Ad esempio, l'espressione

$$(x * y + z) * (w - k)$$

può essere rappresentata dal seguente albero:



## Attraversamento di alberi binari

Nel caso delle liste abbiamo esaminato una semplice strategia di scansione che consiste nella visita di tutti i nodi di una lista, dal primo all'ultimo. Per visitare tutti i nodi di un albero o, in altre parole, per *attraversarlo* per intero, si possono definire strategie differenti. Tali strategie possono essere esplicite a partire dalla definizione ricorsiva di albero.

In particolare le tre strategie fondamentali per attraversare tutti i nodi di un albero  $T$  con radice  $r$ , sottoalbero sinistro  $T_{sx}$  e sottoalbero destro  $T_{dx}$ , sono:

### Attraversamento in ordine anticipato o *preordine*.

Si visita prima la radice  $r$ , poi il sottoalbero sinistro  $T_{sx}$ , e infine il sottoalbero destro  $T_{dx}$ .

### Attraversamento in ordine simmetrico o *inordine*.

Si visita prima il sottoalbero sinistro  $T_{sx}$ , poi la radice  $r$ , e infine il sottoalbero destro  $T_{dx}$ .

### Attraversamento in ordine posticipato o *postordine*.

Si visita prima il sottoalbero sinistro  $T_{sx}$ , poi il sottoalbero destro  $T_{dx}$ , e infine la radice  $r$ .

Ad esempio, visitando nei tre ordini l'albero rappresentato nell'ultima figura si ottiene:

**Ordine anticipato**  $* + * x y z - w k$

**Ordine simmetrico**  $x * y + z * w = k$

**Ordine posticipato**  $x y * z + w k = *$

Si noti che nella sequenza ottenuta leggendo l'espressione in ordine simmetrico (notazione *infissa*) c'è perdita d'informazione, in quanto mancano le parentesi.

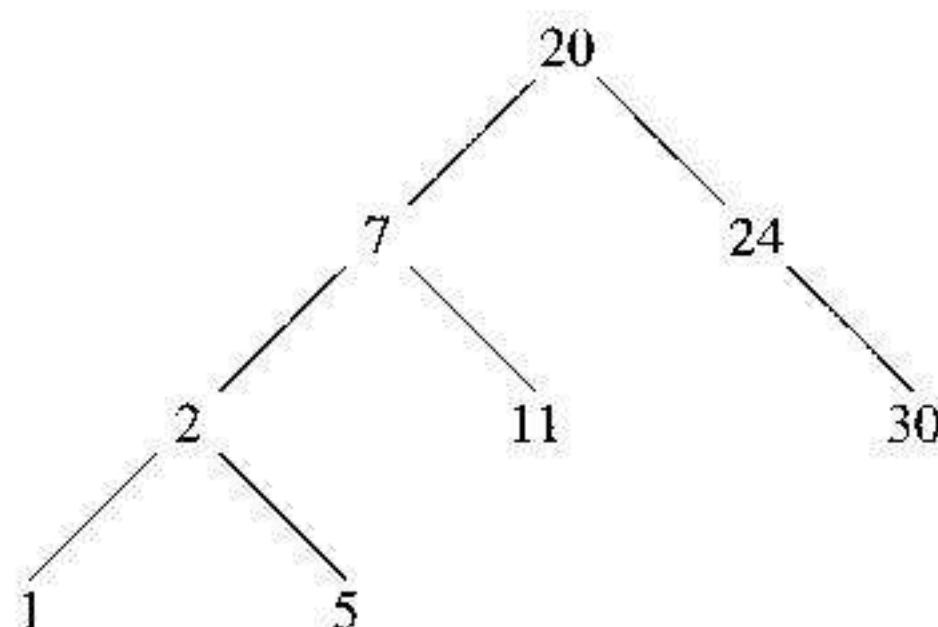
Al contrario, le sequenze ottenute mediante le visite in ordine anticipato e in ordine posticipato corrispondono, rispettivamente, alla notazione *prefissa* e alla notazione *postfissa*, tramite le quali è possibile rappresentare un'espressione aritmetica *senza utilizzare le parentesi*.

## 12.5 Alberi di ricerca

Gli alberi binari sono spesso usati per rappresentare insiemi di dati ordinati in base a una chiave. A questo scopo si considerano alberi particolari, detti *alberi di ricerca*:

**Definizione** Un albero di ricerca è un albero binario in cui per ogni nodo  $x$  tutti i valori contenuti nel sottoalbero sinistro di  $x$  sono minori del valore contenuto in  $x$ , mentre tutti i valori contenuti nel sottoalbero destro di  $x$  sono maggiori (o uguali, nel caso si permettano valori ripetuti) del valore contenuto in  $x$ .

Questa figura rappresenta un albero di ricerca contenente numeri interi:



Si osservi che, visitando un albero di ricerca in ordine simmetrico, si attraversano nell'ordine:

- i nodi del sottoalbero sinistro, cioè tutti i nodi i cui valori sono minori del valore della radice;
- la radice;
- i nodi del sottoalbero destro, cioè tutti i nodi i cui valori sono maggiori del valore della radice.

Pertanto la visita in ordine simmetrico dei nodi di un albero di ricerca produce un elenco ordinato dei valori memorizzati nell'albero.

I valori incontrati visitando in ordine simmetrico l'albero nella figura precedente sono:

- (1) i valori del sottoalbero sinistro della radice;
- (2) il valore della radice, cioè 20;
- (3) i valori del sottoalbero destro della radice.

Applicando ricorsivamente ai sottoalberi lo stesso metodo di visita si ottiene la sequenza dei valori 1 2 5 7 11 20 24 30, cioè la sequenza ordinata dei valori memorizzati nell'albero.

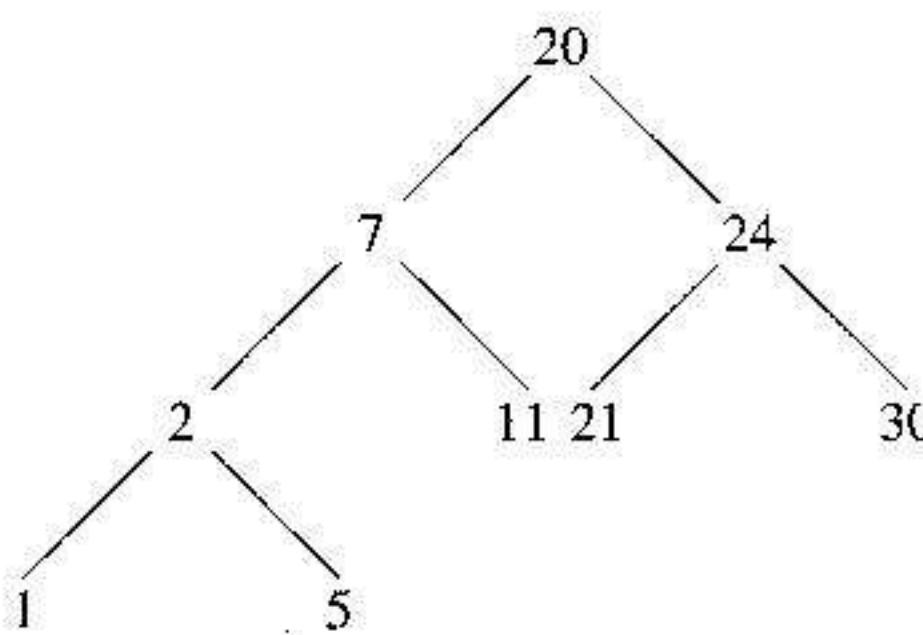
Vediamo come si può costruire un albero di ricerca. Osserviamo anzitutto che l'albero vuoto è un caso banale di albero di ricerca. Inoltre, dato un albero di ricerca, per inserirvi un nuovo elemento si può utilizzare la seguente strategia, basata sulla definizione ricorsiva di albero:

- se l'albero è vuoto, si inserisce l'elemento immediatamente, ottenendo un albero con un unico nodo;
- se l'albero è costituito da una radice e da due sottoalberi, si confronta il valore da inserire con il valore contenuto nella radice:
  - se il valore da inserire è minore del valore della radice, si effettua l'inserimento ricorsivamente a sinistra;
  - altrimenti si effettua l'inserimento a destra.

Ad esempio, volendo inserire il valore 21 nell'albero raffigurato in precedenza, effettuiamo i seguenti passi a partire dalla radice:

- confrontiamo il valore della radice, cioè 20, con il valore da inserire; poiché questo è maggiore, procediamo ricorsivamente sul sottoalbero destro;
- confrontiamo il valore della radice del sottoalbero considerato, cioè 24, con il valore da inserire; poiché quest'ultimo è minore, procediamo ricorsivamente sul sottoalbero sinistro del nodo considerato (cioè nel sottoalbero sinistro del nodo contenente 24);
- il sottoalbero considerato è ora vuoto; procediamo dunque all'inserimento.

Effettuato l'inserimento, l'albero diventa:



## 12.6 Implementazione degli alberi di ricerca in Java

Presentiamo ora un'implementazione degli alberi di ricerca in Java. In particolare presentiamo una classe `Albero` per rappresentare alberi di ricerca i cui nodi contengano oggetti generici. Come nel caso della classe `ListaOrdinata`, al fine di disporre di un metodo per il confronto degli oggetti da memorizzare nell'albero, l'unica richiesta che facciamo è che gli oggetti appartengano a una classe che implementi l'interfaccia `Comparable`.

Abbiamo detto che un albero è la struttura vuota, oppure un nodo, la radice, cui sono associate due strutture ad albero: il sottoalbero sinistro e il sottoalbero destro. La rappresentazione che utilizziamo rispecchia direttamente questa definizione. Un oggetto della classe `Albero` ha un campo `a` che è un riferimento a un oggetto di una classe `NodoAlbero`. Se il campo `a` contiene `null`, l'albero rappresentato è quello vuoto, altrimenti `a` si riferisce alla radice dell'albero. Un oggetto della classe `NodoAlbero` contiene a sua volta un campo `dato`, destinato a contenere il riferimento all'oggetto da memorizzare nel nodo, più due campi `sx` e `dx` di tipo `Albero<E>`, che sono, rispettivamente, i riferimenti ai sottoalberi sinistro e destro:

```
public class Albero<E extends Comparable<? super E>> {

 private NodoAlbero a;

 public Albero() {
 a = null;
 }

 private class NodoAlbero {
 E dato;
 Albero<E> sx, dx;
 }

 ...metodi...
}
```

Il costruttore scritto sopra costruisce un albero vuoto. Per visualizzare il contenuto dell'albero scriviamo un metodo `void visualizza()` basato sulla visita in ordine simmetrico. Se l'albero è vuoto, non c'è nulla da visualizzare; se, al contrario, l'albero non è vuoto, si devono effettuare le seguenti operazioni:

- si richiede al sottoalbero sinistro della radice di visualizzare il proprio contenuto;
- si visualizza il contenuto della radice;
- si richiede al sottoalbero destro della radice di visualizzare il proprio contenuto.

Per richiedere al sottoalbero sinistro e al sottoalbero destro di visualizzare il proprio contenuto si domanderà l'esecuzione ricorsiva del metodo `visualizza`:

```

public void visualizza() {
 if (a != null) {
 a.sx.visualizza();
 System.out.println(a.dato);
 a.dx.visualizza();
 }
}

```

L'introduzione di un metodo che effettui direttamente l'output, come `visualizza`, non è molto elegante. Presentiamo ora un metodo `toString` che restituisce una stringa che rappresenta il contenuto dell'albero. Per scrivere il metodo ci basiamo su questo schema ricorsivo.

- L'albero vuoto è rappresentato dalla stringa vuota.
- Se l'albero non è vuoto, `c sinistra` e `c destra` sono due stringhe che rappresentano, rispettivamente, il sottoalbero sinistro e destro, e `c centro` è una stringa che rappresenta l'oggetto memorizzato nella radice, in base all'ordine simmetrico l'intero albero è rappresentato dalla concatenazione delle stringhe `sinistra`, `centro` e `destra`.

Il codice del metodo, riportato sotto, si basa su una selezione nei cui due rami sono considerati i casi indicati sopra (il parametro del metodo viene utilizzato per separare tra loro le varie stringhe):

```

public String toString(String separatore) {
 if (a != null) {
 String sinistra = a.sx.toString(separatore);
 String centro = a.dato.toString() + separatore;
 String destra = a.dx.toString(separatore);
 return sinistra + centro + destra;
 } else
 return "";
}

```

Scriviamo ora un metodo `void inserisci(E x)` che effettui l'inserimento di un elemento nell'albero. Anche in questo caso seguiamo la definizione ricorsiva di albero: se l'albero è vuoto, effettuiamo immediatamente l'inserimento, altrimenti lo effettuiamo nel sottoalbero sinistro o destro in base al risultato del confronto tra l'oggetto da inserire e quello memorizzato nella radice:

```

public void inserisci(E x) {
 if (a == null) {
 inserisci qui un nuovo nodo contenente x,
 con sottoalberi sinistro e destro vuoti
 } else if (x è minore di a.dato)
 inserisci x nel sottoalbero sinistro
}

```

```

 else
 inserisci x nel sottoalbero destro
}

```

Per inserire x nel sottoalbero sinistro chiediamo al sottoalbero sinistro, riferito da a.sx, di eseguire il proprio metodo `inserisci`:

```
a.sx.inserisci(x);
```

Si può effettuare l'inserimento a destra in maniera analoga. Nel caso in cui a sia null, occorre prima di tutto creare un nuovo nodo e farlo riferire da a:

```
a = new NodoAlbero();
```

In questo nodo memorizziamo il riferimento all'oggetto da inserire:

```
a.dato = x;
```

Infine occorre associare al nodo un sottoalbero sinistro e un sottoalbero destro. Entrambi devono essere vuoti. A tale scopo invochiamo il costruttore della classe `Albero` (precisando che il tipo argomento è sempre E):

```
a.sx = new Albero<E>();
a.dx = new Albero<E>();
```

Ecco il codice completo del metodo di inserimento:

```

public void inserisci(E x) {
 if (a == null) {
 a = new NodoAlbero();
 a.dato = x;
 a.sx = new Albero<E>();
 a.dx = new Albero<E>();
 } else if (x.compareTo(a.dato) < 0)
 a.sx.inserisci(x);
 else
 a.dx.inserisci(x);
}

```

Con la stessa tecnica possiamo sviluppare un metodo per la ricerca di un oggetto nell'albero: ricevendo tramite il parametro l'oggetto da cercare, il metodo restituisce un riferimento a un oggetto uguale, se presente nell'albero, null in caso contrario:

```

public E trova(E x) {
 if (a == null)
 return null;
 else if (x.compareTo(a.dato) < 0)

```

```

 return a.sx.trova(x);
 else if (x.compareTo(a.dato) > 0)
 return a.dx.trova(x);
 else
 return a.dato;
}

```

Si osservi che il metodo restituisce `null` quando l'albero è vuoto. Tale situazione viene raggiunta nella sequenza delle chiamate ricorsive se l'elemento cercato non è presente nell'albero.

Il metodo `inserisci`, presentato sopra, effettua l'inserimento anche se l'oggetto è già presente nell'albero (in questo caso l'inserimento avviene a destra). In molte applicazioni, se l'oggetto è già presente, l'inserimento non dev'essere effettuato. In alternativa può essere necessario svolgere altre operazioni sull'oggetto già memorizzato nell'albero (vedremo una situazione di questo tipo più avanti). A tale scopo è utile scrivere un metodo di *ricerca con inserimento*. Il metodo ricerca l'oggetto nell'albero:

- se è presente, il metodo restituisce un riferimento a esso (grazie a questo riferimento il chiamante può modificare lo stato dell'oggetto trovato nell'albero);
- se non è presente, il metodo inserisce l'oggetto nell'albero e restituisce il riferimento `null`.

La struttura del metodo è abbastanza simile a quella dei metodi `trova` e `inserisci`:

```

public E trovaEInserisci(E x) {
 if (a == null) {
 a = new NodoAlbero();
 a.dato = x;
 a.sx = new Albero<E>();
 a.dx = new Albero<E>();
 return null;
 } else if (x.compareTo(a.dato) < 0)
 return a.sx.trovaEInserisci(x);
 else if (x.compareTo(a.dato) > 0)
 return a.dx.trovaEInserisci(x);
 else
 return a.dato;
}

```

Segue il codice della classe, completo dei metodi presentati sopra e di alcuni metodi aggiuntivi:

```

public class Albero<E extends Comparable<? super E>> {
 private NodoAlbero a;

 public Albero() {
 a = null;
 }
}

```

```
}

private class NodoAlbero {
 E dato;
 Albero<E> sx, dx;
}

public String toString(String separatore) {
 if (a != null) {
 String sinistra = a.sx.toString(separatore);
 String centro = a.dato.toString() + separatore;
 String destra = a.dx.toString(separatore);
 return sinistra + centro + destra;
 } else
 return "";
}

public void visualizza() {
 if (a != null) {
 a.sx.visualizza();
 System.out.println(a.dato);
 a.dx.visualizza();
 }
}

public void inserisci(E x) {
 if (a == null) {
 a = new NodoAlbero();
 a.dato = x;
 a.sx = new Albero<E>();
 a.dx = new Albero<E>();
 } else if (x.compareTo(a.dato) < 0)
 a.sx.inserisci(x);
 else
 a.dx.inserisci(x);
}

public E trova(E x) {
 if (a == null)
 return null;
 else if (x.compareTo(a.dato) < 0)
 return a.sx.trova(x);
```

```

 else if (x.compareTo(a.dato) > 0)
 return a.dx.trova(x);
 else
 return a.dato;
}

public E trovaEInserisci(E x) {
 if (a == null) {
 a = new NodoAlbero();
 a.dato = x;
 a.sx = new Albero<E>();
 a.dx = new Albero<E>();
 return null;
 } else if (x.compareTo(a.dato) < 0)
 return a.sx.trovaEInserisci(x);
 else if (x.compareTo(a.dato) > 0)
 return a.dx.trovaEInserisci(x);
 else
 return a.dato;
}

public boolean vuoto() {
 return a == null;
}
}

```

Ecco il testo di una semplice applicazione di prova per alcuni metodi:

```

import prog.io.*;

public class ProvaAlbero {
 private static ConsoleInputManager in = new ConsoleInputManager();
 private static ConsoleOutputManager out = new ConsoleOutputManager();

 public static void main(String[] args) {
 int scelta;
 String parola;
 Albero<String> albero = new Albero<String>();
 while ((scelta = menu()) != 0)
 switch (scelta) {
 case 1:
 parola = in.readLine("Stringa da inserire? ");
 albero.inserisci(parola);

```

```
 break;
 case 2:
 if (albero.vuoto())
 out.println("Non è stata memorizzata alcuna stringa");
 else
 out.println(albero.toString("\n"));
 break;
 case 3:
 parola = in.readLine("Stringa da cercare? ");
 if (albero.trova(parola) == null)
 out.println("La stringa " + parola + " non è presente");
 else
 out.println("La stringa " + parola + " è presente");
 break;
 case 0:
 break;
 default:
 out.println("Scelta non valida");
 break;
 }
}

public static int menu() {
 out.println("\nScelte disponibili:\n");
 out.println("1. Inserimento di una nuova stringa");
 out.println("2. Elenco delle stringhe memorizzate");
 out.println("3. Ricerca di una stringa");
 out.println();
 out.println("0. Uscita");
 out.println();
 int scelta = in.readInt("Scelta? ");
 return scelta;
}
}
```

## 12.7 Esempio: elenco alfabetico di stringhe

Vogliamo costruire una classe `ContaOccorrenze` che legga un elenco di stringhe inserite dall'utente e le riscriva in ordine alfabetico, indicando per ciascuna stringa quante volte è stata inserita. Se ad esempio l'utente inserisce:

```
topo
cane
elefante
pipistrello
topo
gatto
```

l'output prodotto dovrà essere della forma:

```
cane 1
elefante 1
gatto 1
pipistrello 1
topo 2
```

L'idea è quella di utilizzare un albero di ricerca, ordinato secondo l'ordine lessicografico, per memorizzare le stringhe e i relativi contatori. La classe `ContaOccorrenze` è costituita da un unico metodo `main` in cui, una volta aperti i canali per la lettura e la scrittura, vengono effettuate le seguenti operazioni:

```
crea un albero inizialmente vuoto
leggi una stringa
while (la stringa non indica la fine dell'inserimento) {
 if (la stringa non è presente nell'albero)
 inseriscila con contatore uguale a 1
 else
 incrementane il contatore
 leggi una stringa
}
visualizza l'elenco delle stringhe con i relativi contatori
```

Per memorizzare le stringhe con i relativi contatori possiamo ricorrere alle classi `Occorrenza` e `OccorrenzaOrdinata`, presentate nel Paragrafo 6.11 e nell'Esercizio 6.32. Ricordiamo che queste classi permettono di contare le occorrenze di oggetti di un tipo parametro `E`. Nel caso della classe `Occorrenza` il tipo `E` può essere qualunque, nel caso di `OccorrenzaOrdinata` si richiede che `E` implementi l'interfaccia `Comparable`. A sua volta, `OccorrenzaOrdinata` implementa `Comparable`: l'ordine definito sugli oggetti di tipo `OccorrenzaOrdinata` deriva dall'ordine definito sugli oggetti del tipo `E`. Ad esempio, per `OccorrenzaOrdinata<String>`, l'ordine è quello alfabetico tra stringhe.

Sviluppiamo ora la classe `ContaOccorrenze` sulla base dello schema riportato in precedenza. Per la creazione dell'albero vuoto, non facciamo altro che richiamare il costruttore della classe `Albero` e memorizzare il riferimento così ottenuto in una variabile di nome `albero`. L'operazione è elementare, la sintassi è invece prolissa in quanto dobbiamo specificare il tipo argomento di `Albero`:

```
Albero<OccorrenzaOrdinata<String>> albero =
 new Albero<OccorrenzaOrdinata<String>>();
```

Sviluppiamo ora il corpo del ciclo. Supponiamo che a ogni iterazione la stringa letta si trovi in una variabile `riga` di tipo `String`. Per il test all'interno del ciclo procediamo come segue. Costruiamo prima di tutto un'occorrenza per `riga`. Richiamiamo quindi il metodo `trovaEInserisci` dell'albero fornendo come argomento tale occorrenza. Se `riga` non era già presente, il metodo `trovaEInserisci` effettua l'inserimento e restituisce `null`; altrimenti il metodo `trovaEInserisci` restituisce il riferimento all'occorrenza già presente nell'albero associata a `riga`. In questo caso chiediamo dunque a tale descrittore di eseguire il proprio metodo `incrementa`:

```
OccorrenzaOrdinata<String> d = new OccorrenzaOrdinata<String>(riga);
OccorrenzaOrdinata<String> ris = albero.trovaEInserisci(d);
if (ris != null)
 ris.incrementa();
```

Il ciclo viene ripetuto fintantoché non viene inserita la stringa vuota. Infine, per visualizzare le stringhe con i relativi contatori è sufficiente richiamare il metodo `toString` dell'oggetto riferito da `albero`. Ricordiamo che tale metodo fornisce la stringa ottenuta effettuando la visita dell'albero in ordine simmetrico. Ecco il codice della classe `ContaOccorrenze` così sviluppata:

```
import prog.utili.OccorrenzaOrdinata;
import prog.io.*;

public class ContaOccorrenze {
 public static void main(String[] args) {
 ConsoleInputManager in = new ConsoleInputManager();
 ConsoleOutputManager out = new ConsoleOutputManager();

 Albero<OccorrenzaOrdinata<String>> albero =
 new Albero<OccorrenzaOrdinata<String>>();
 out.println("Inserire le stringhe");
 String riga = in.readLine();
 while (!riga.equals("")) {
 OccorrenzaOrdinata<String> d = new OccorrenzaOrdinata<String>(riga);
 OccorrenzaOrdinata<String> ris = albero.trovaEInserisci(d);
 if (ris != null)
```

```

 ris.incrementa(); //se c'era già incrementa
 //il numero di occorrenze
 riga = in.readLine();
}
out.println();
out.println(albero.toString("\n"));
}
}

```

## Esercizi

- 12.14 Considerate la seguente espressione aritmetica:

$$3 + 5 * (4 - 2) + 11 * 2 + (1 * 6)$$

Disegnate l'albero binario che la rappresenta. Riscrivete poi l'espressione in notazione prefissa e in notazione postfissa.

- 12.15 Disegnate l'albero di ricerca ottenuto inserendo uno dopo l'altro in un albero inizialmente vuoto i numeri 15 12 11 16 10 3 99 87 14 13.

Scrivete gli output prodotti visitando tale albero nei tre ordini: anticipato, simmetrico e posticipato.

- 12.16 Disegnate l'albero di ricerca ottenuto inserendo uno dopo l'altro in un albero inizialmente vuoto i numeri 20 11 18 16 17 3 30 4 19 25.

Scrivete gli output prodotti visitando tale albero nei tre ordini: anticipato, simmetrico e posticipato.

- 12.17 Disegnate l'albero di ricerca ottenuto inserendo uno dopo l'altro in un albero inizialmente vuoto i numeri 14 11 16 15 17 23 13 40 22 25.

Scrivete gli output prodotti visitando tale albero nei tre ordini: anticipato, simmetrico e posticipato.

- 12.18 Ampliate la classe `Albero` scrivendo un metodo che restituisca il numero di foglie dell'albero.

- 12.19 Ampliate la classe `Albero` scrivendo un metodo che restituisca la profondità dell'albero.

- 12.20 Dimostrate che in un albero binario contenente  $n$  nodi ci sono  $n + 1$  riferimenti uguali a null. Che cosa si può dire nel caso degli alberi ternari?

- 12.21 Svolgete l'Esercizio 6.32 utilizzando una struttura ad albero. Ispiratevi all'applicazione `ContaOccorrenze`.

12.22 Il metodo `toString` della classe `Albero` permette di ottenere una stringa mediante l’attraversamento in ordine simmetrico. Può essere utile disporre di un elenco degli oggetti presenti nell’albero, ottenuti secondo l’ordine simmetrico. Tale elenco può essere fornito sotto forma di iteratore. Ampliate la classe `Albero` in modo che implementi l’interfaccia `Iterable`. In particolare il metodo `iterator`, privo di argomenti, previsto dall’interfaccia dovrà restituire l’elenco di tutti gli elementi presenti nell’albero ottenuti secondo la visita in ordine simmetrico. Un modo semplice (ma non molto efficiente) per implementare il metodo `iterator` consiste nel costruire un oggetto di tipo `Sequenza` contenente gli elementi ottenuti mediante l’attraversamento in ordine simmetrico e nel richiamare poi il metodo `iterator` della classe `Sequenza` per ricavare l’iteratore.

12.23 Ampliate la classe `Albero` con i seguenti tre metodi:

- `public Iterator<E> elencoVisitaAnticipata()`
- `public Iterator<E> elencoVisitaSimmetrica()`
- `public Iterator<E> elencoVisitaPosticipata()`

Tutti e tre questi metodi devono restituire un iteratore contenente gli oggetti memorizzati nell’albero. Nel caso del primo metodo gli elementi devono comparire nell’iteratore nell’ordine corrispondente alla visita anticipata dell’albero; nel caso del secondo metodo devono comparire nell’ordine corrispondente alla visita simmetrica; nel caso del terzo metodo devono comparire nell’ordine corrispondente alla visita posticipata.

Infine modificate la classe in modo da consentire ai suoi utilizzatori di stabilire (mediante l’invocazione di un opportuno metodo statico) l’ordine in cui devono comparire gli oggetti memorizzati nell’albero nell’iteratore restituito del metodo `iterator` definito dall’interfaccia `Iterable`. Scrivete un’applicazione di prova per i nuovi metodi della classe.

12.24 Modificate l’applicazione scritta per l’Esercizio 12.21 in modo che, oltre al numero di occorrenze, per ogni parola vengano comunicati anche i numeri delle righe in cui la parola compare (si contino le righe progressivamente a partire da 1). In questo caso è opportuno scrivere una classe che estenda `OccorrenzaOrdinata<String>`, nella quale venga associata a ogni oggetto di tipo `OccorrenzaOrdinata<String>` una coda o una `Sequenza` che memorizzi i numeri delle righe in cui la stringa considerata compare.

Ad esempio, se il file di testo considerato è il seguente:

Nel mezzo del cammin di nostra vita  
mi ritrovai per una selva oscura  
che' la diritta via era smarrita.  
Ahi quanto a dir qual era e' cosa dura  
esta selva selvaggia e aspra e forte  
che nel pensier rinnova la paura!

L’output prodotto sul file di esempio dell’esercizio precedente dovrà essere:

|         |                    |     |
|---------|--------------------|-----|
| Ahi     | Numero occorrenze: | 1   |
|         | Righe:             | 4   |
| Nel     | Numero occorrenze: | 1   |
|         | Righe:             | 1   |
| a       | Numero occorrenze: | 1   |
|         | Righe:             | 4   |
| aspra   | Numero occorrenze: | 1   |
|         | Righe:             | 5   |
| cammin  | Numero occorrenze: | 1   |
|         | Righe:             | 1   |
| che     | Numero occorrenze: | 2   |
|         | Righe:             | 3 6 |
| cosa    | Numero occorrenze: | 1   |
|         | Righe:             | 4   |
| del     | Numero occorrenze: | 1   |
|         | Righe:             | 1   |
| di      | Numero occorrenze: | 1   |
|         | Righe:             | 1   |
| dir     | Numero occorrenze: | 1   |
|         | Righe:             | 4   |
| diritta | Numero occorrenze: | 1   |
|         | Righe:             | 3   |
| dura    | Numero occorrenze: | 1   |
|         | Righe:             | 4   |
| e       | Numero occorrenze: | 3   |
|         | Righe:             | 4 5 |
| era     | Numero occorrenze: | 2   |
|         | Righe:             | 3 4 |
| esta    | Numero occorrenze: | 1   |
|         | Righe:             | 5   |
| forte   | Numero occorrenze: | 1   |
|         | Righe:             | 5   |
| la      | Numero occorrenze: | 2   |
|         | Righe:             | 3 6 |
| mezzo   | Numero occorrenze: | 1   |
|         | Righe:             | 1   |
| mi      | Numero occorrenze: | 1   |
|         | Righe:             | 2   |
| nel     | Numero occorrenze: | 1   |
|         | Righe:             | 6   |
| nostra  | Numero occorrenze: | 1   |
|         | Righe:             | 1   |

|           |                    |     |
|-----------|--------------------|-----|
| oscura    | Numero occorrenze: | 1   |
|           | Righe:             | 2   |
| paura     | Numero occorrenze: | 1   |
|           | Righe:             | 6   |
| pensier   | Numero occorrenze: | 1   |
|           | Righe:             | 6   |
| per       | Numero occorrenze: | 1   |
|           | Righe:             | 2   |
| qual      | Numero occorrenze: | 1   |
|           | Righe:             | 4   |
| quanto    | Numero occorrenze: | 1   |
|           | Righe:             | 4   |
| rinova    | Numero occorrenze: | 1   |
|           | Righe:             | 6   |
| ritrovai  | Numero occorrenze: | 1   |
|           | Righe:             | 2   |
| selva     | Numero occorrenze: | 2   |
|           | Righe:             | 2 5 |
| selvaggia | Numero occorrenze: | 1   |
|           | Righe:             | 5   |
| smarrita  | Numero occorrenze: | 1   |
|           | Righe:             | 3   |
| una       | Numero occorrenze: | 1   |
|           | Righe:             | 2   |
| via       | Numero occorrenze: | 1   |
|           | Righe:             | 3   |
| vita      | Numero occorrenze: | 1   |
|           | Righe:             | 1   |

12.25 Definite una classe `AlberoInteri` per rappresentare alberi di ricerca contenenti numeri interi. Ponete nella classe metodi analoghi a quelli della classe `Albero`. Scrivete inoltre i seguenti metodi:

- (1) un metodo che restituisca la somma dei valori contenuti nei nodi dell'albero;
- (2) un metodo che restituisca la somma dei valori contenuti nelle foglie dell'albero;
- (3) un metodo che restituisca la media dei valori contenuti nell'albero (nel caso l'albero sia vuoto, il metodo deve sollevare un'eccezione);
- (4) un metodo che restituisca il più grande valore contenuto nell'albero (nel caso l'albero sia vuoto, il metodo deve sollevare un'eccezione);
- (5) un metodo che restituisca il più piccolo valore contenuto nell'albero (nel caso l'albero sia vuoto, il metodo deve sollevare un'eccezione).

## 12.8 Ulteriori esempi sulle liste

Negli esempi seguenti consideriamo una classe `ListaInteri` per rappresentare liste di numeri interi.<sup>5</sup> Supponiamo che la classe sia definita così:

```
public class ListaInteri {
 private NodoLista inizio;

 private class NodoLista {
 int dato;
 NodoLista pros;
 }

 public ListaInteri() {
 inizio = null;
 }

 ...metodi...
}
```

### Scambio del valore minimo con il valore massimo

Vogliamo costruire un metodo della classe `ListaInteri` che trasformi la lista scambiando il valore minimo con il valore massimo. Ad esempio la lista contenente nell'ordine 2 8 1 6 4 13 2 deve essere trasformata nella lista che contiene nell'ordine 2 8 13 6 4 1 2. Per semplicità supponiamo che il valore minimo e massimo compaiano nella lista solo una volta.

Scriviamo prima di tutto l'intestazione del metodo:

```
public void scambiaMinMax()
```

Il metodo può essere suddiviso in due fasi fondamentali: nella prima si scandisce la lista cercando il valore massimo e il valore minimo, nella seconda si effettua lo scambio. Per semplificare la fase di scambio, oltre ai valori del massimo e del minimo, nella fase di ricerca è utile memorizzare anche i riferimenti ai nodi che li contengono.

Dichiariamo dunque due variabili `max` e `min` di tipo `int` destinate a contenere il valore del massimo e del minimo, e due variabili `pmax` e `pmin` di tipo `NodoLista` destinate a contenere i riferimenti ai nodi contenenti il massimo e il minimo. Nel caso di lista non vuota, inizializziamo tali variabili con i valori corrispondenti al primo nodo della lista:

```
pmin = pmax = inizio;
min = max = inizio.dato;
```

---

<sup>5</sup> Questo paragrafo e il successivo mostrano ulteriori esempi relativi alla manipolazione di liste e alberi. Poiché l'obiettivo è concentrarsi su queste strutture piuttosto che sul linguaggio, per semplicità consideriamo versioni non generiche di tali strutture dati.

La ricerca avviene partendo dal secondo elemento e confrontando di volta in volta i valori incontrati con `max` e `min`.

```
p = inizio.pros;
while (p != null) {
 if (p.dato < min) {
 pmin = p;
 min = p.dato;
 }
 if (p.dato > max) {
 pmax = p;
 max = p.dato;
 }
 p = p.pros;
}
```

Al termine della ricerca basta porre il valore `min` nel nodo riferito da `pmax` e il valore `max` nel nodo riferito da `pmin`:

```
pmin.dato = max;
pmax.dato = min;
```

Si noti che, se il valore minimo o massimo sono presenti più di una volta, con questo procedimento vengono scambiate le prime occorrenze dei due valori.

Nel caso di lista vuota, il metodo non dovrà operare alcuna trasformazione. Pertanto racchiusiamo il codice precedente come primo blocco di un'istruzione `if`:

```
public void scambiaMinMax() {
 int min, max;
 NodoLista p, pmin, pmax;

 if (inizio != null) {
 pmin = pmax = inizio;
 min = max = inizio.dato;
 p = inizio.pros;
 while (p != null) {
 if (p.dato < min) {
 pmin = p;
 min = p.dato;
 }
 if (p.dato > max) {
 pmax = p;
 max = p.dato;
 }
 p = p.pros;
 }
 pmin.dato = max;
 pmax.dato = min;
 }
}
```

```

 }
 pmin.dato = max;
 pmax.dato = min;
 }
}

```

## Calcolo della media

Costruiamo ora un metodo che restituisca la media dei valori contenuti nella lista. Poiché la media è un numero reale, l'intestazione sarà:

```
public double media()
```

Il metodo scandisce la lista dal primo all'ultimo elemento mediante un ciclo, e calcola la somma e il numero di elementi presenti. Alla fine il metodo restituisce il risultato della divisione tra la somma e il numero di elementi (uno degli operandi viene forzato a `double` in modo che venga effettuata la divisione fra `double`):

```

public double media() {
 int somma = 0, nelem = 0;
 for (NodoLista p = inizio; p != null; p = p.pros) {
 somma += p.dato;
 nelem++;
 }
 return somma / (double) nelem;
}

```

Si noti che, in caso di lista vuota, il metodo tenta di effettuare una divisione per zero. Nel caso di divisione tra `double`, questo restituisce il valore `NaN` (*Not a Number*). Modifichiamo il metodo in modo che, in questa situazione, sollevi un nuovo tipo di eccezione, definita mediante la seguente classe:

```

public class ListaInteriException extends RuntimeException {

 public ListaInteriException(String s) {
 super(s);
 }
}

```

Ecco il metodo modificato:

```

public double media2() {
 int somma = 0, nelem = 0;
 for (NodoLista p = inizio; p != null; p = p.pros) {
 somma += p.dato;
 }
}

```

```

 nelem++;
 }
 if (nelem == 0)
 throw new ListaInteriException("Impossibile calcolare " +
 "la media di una lista vuota");
 else
 return somma / (double) nelem;
}

```

## 12.9 Ulteriori esempi sugli alberi

Nei prossimi esempi faremo riferimento alla seguente classe, che rappresenta alberi binari di interi:

```

public class AlberoInteri {

 private NodoAlbero a;

 public AlberoInteri() {
 a = null;
 }

 private class NodoAlbero {
 int dato;
 AlberoInteri sx, dx;
 }

 ...metodi...
}

```

I metodi presentati verranno sviluppati alla luce della definizione ricorsiva di albero: per ogni metodo dovremo decidere le azioni da compiere in caso di albero vuoto e in caso di albero formato da un nodo, la radice, e da due sottoalberi.

### Calcolo della somma dei valori contenuti nell'albero

Vogliamo costruire un metodo che restituisca come risultato la somma dei valori contenuti nei nodi dell'albero. L'intestazione sarà:

```
public int somma()
```

Basandoci sulla definizione ricorsiva di albero osserviamo che:

- l'albero vuoto ha somma zero;

- nel caso di un albero non vuoto, la somma è data dalla somma del sottoalbero sinistro più la somma del sottoalbero destro più il valore contenuto nella radice.

La somma del sottoalbero sinistro e quella del sottoalbero destro possono essere ottenute eseguendo (ricorsivamente) i metodi `somma` dei due sottoalberi.

In base al precedente schema otteniamo il seguente codice:

```
public int somma() {
 if (a == null)
 return 0;
 else {
 int sommasx = a.sx.somma();
 int sommadx = a.dx.somma();
 return sommasx + sommadx + a.dato;
 }
}
```

Supponiamo ora di voler calcolare la somma dei soli valori *pari* presenti nell'albero. A questo scopo costruiamo un metodo

```
public int sommaPari()
```

Riferendoci ancora alla definizione ricorsiva osserviamo che:

- per l'albero vuoto il risultato è zero;
- nel caso di un albero non vuoto, il risultato si ottiene sommando i risultati dei sottoalberi sinistro e destro ai quali, solo nel caso di radice contenente un valore pari, va aggiunto il valore della radice;

```
public int sommaPari() {
 if (a == null)
 return 0;
 else {
 int sommasx = a.sx.sommaPari();
 int sommadx = a.dx.sommaPari();
 if (a.dato % 2 == 0)
 return sommasx + sommadx + a.dato;
 else
 return sommasx + sommadx;
 }
}
```

## Calcolo del numero dei nodi

Il numero dei nodi può essere calcolato osservando che:

- il numero dei nodi di un albero vuoto è zero;
- il numero dei nodi di un albero formato da una radice e due sottoalberi è dato dalla somma dei numeri dei nodi dei due sottoalberi più 1 (la radice);

```
public int numeroNodi() {
 if (a == null)
 return 0;
 else {
 int nodisx = a.sx.numeroNodi();
 int nodidx = a.dx.numeroNodi();
 return nodisx + nodidx + 1;
 }
}
```

## Calcolo del numero di foglie

Ricordiamo che una foglia di un albero binario è un nodo privo di figli, cioè un nodo i cui sottoalberi sinistro e destro sono vuoti. Possiamo calcolare il numero di foglie osservando che:

- un albero vuoto non ha nodi, e dunque ha zero foglie
- se l'albero è formato da un nodo e due sottoalberi, possono verificarsi i seguenti casi:
  - se ambedue i sottoalberi sono vuoti, il nodo è una foglia; dunque il numero di foglie dell'albero è 1
  - altrimenti il numero complessivo di foglie dell'albero è dato dal numero di foglie del sottoalbero sinistro più il numero di foglie del sottoalbero destro.

In base a questo schema possiamo scrivere il seguente metodo:

```
public int numeroFoglie() {
 if (a == null)
 return 0;
 else if (a.sx.vuoto() && a.dx.vuoto())
 return 1;
 else
 return a.sx.numeroFoglie() + a.dx.numeroFoglie();
}
```

Si può scrivere in modo analogo questo metodo per il calcolo della somma delle foglie contenute in un albero:

```
public int sommaFoglie() {
 if (a == null)
 return 0;
 else if (a.sx.vuoto() && a.dx.vuoto())
 return a.dato;
 else
 return a.sx.sommaFoglie() + a.dx.sommaFoglie();
}
```

### Calcolo della profondità di un albero binario

Ricordiamo che in un albero binario il livello della radice è 1, mentre il livello dei figli di un nodo di livello  $i$  è  $i + 1$ . La profondità dell'albero è il massimo livello dei suoi nodi.

Vogliamo costruire un metodo che restituisca la profondità dell'albero. L'intestazione del metodo sarà:

```
public int profondita()
```

Per scrivere il codice individuiamo una regola ricorsiva per il calcolo della profondità.

- L'*albero vuoto*, essendo privo di nodi, ha profondità zero.
- La profondità dell'albero costituito da una *radice* e due *sottoalberi*,  $T_{sx}$  e  $T_{dx}$ , si ottiene aggiungendo 1 alla maggiore tra le profondità di  $T_{sx}$  e  $T_{dx}$ .

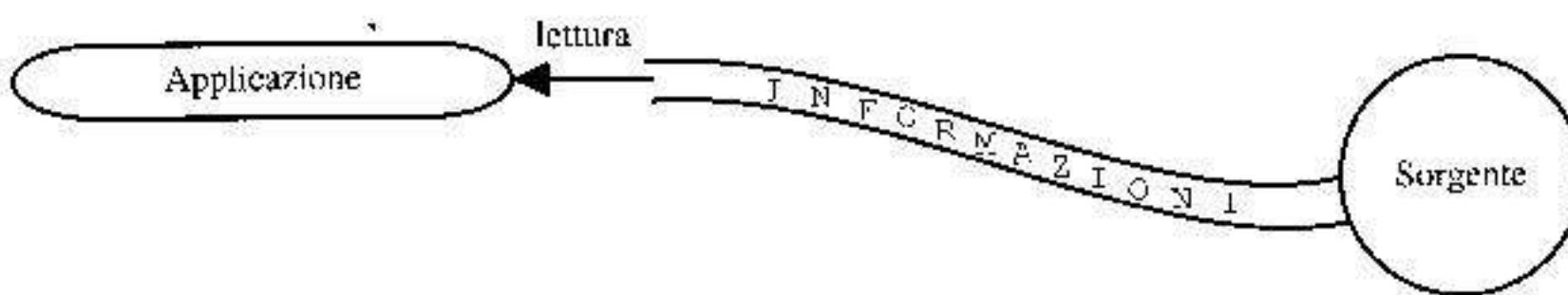
A partire da queste regole è naturale scrivere il seguente metodo:

```
public int profondita() {
 if (a == null)
 return 0;
 else {
 int psx = a.sx.profondita();
 int pdx = a.dx.profondita();
 return psx > pdx ? psx + 1 : pdx + 1;
 }
}
```

# Gli stream

In questo capitolo descriveremo gli aspetti principali del package di input/output `java.io` della libreria standard di Java, evidenziando l'organizzazione delle classi in esso contenute senza pretendere di fornirne una descrizione esaustiva, per la quale rimandiamo alla documentazione. In particolare ci concentreremo sul caso in cui la sorgente e la destinazione dei dati sono file.

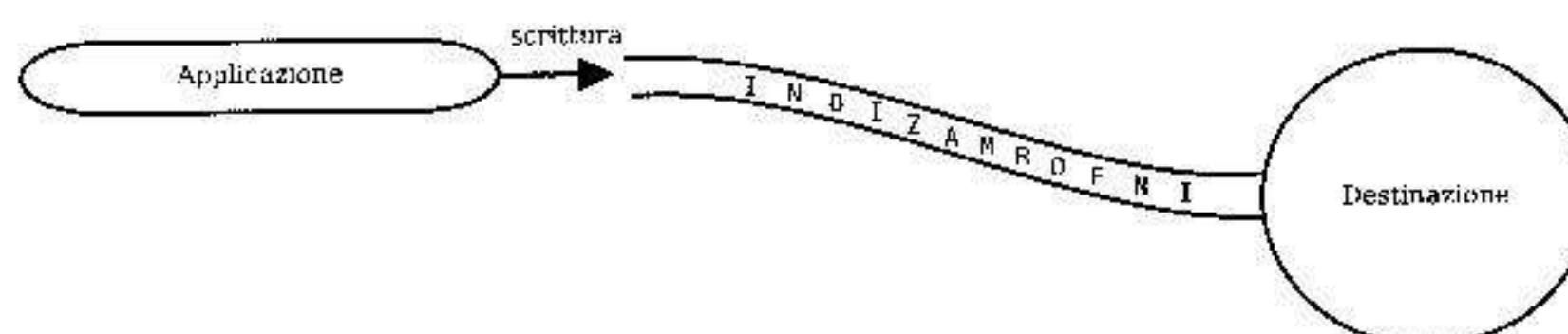
Il package `java.io` descrive l'input/output mediante il concetto di *stream* (*flusso*). Gli stream sono sequenze ordinate di dati che hanno una *sorgente* e una *destinazione*. Per prelevare informazioni da una sorgente, un'applicazione deve aprire uno stream collegato alla sorgente e leggere sequenzialmente, cioè una dopo l'altra, le informazioni in esso contenute:



Il processo di lettura di dati da uno stream può essere sintetizzato come segue:

```
apri lo stream
while (ci sono dati da leggere nello stream)
 leggi un dato dallo stream
chiudi lo stream
```

Analogamente, per inviare dati a una destinazione, un'applicazione deve aprire uno stream collegato con la destinazione e deve scrivere le informazioni su di esso:



Il processo di scrittura è schematizzato nel modo seguente:

```
apri lo stream
while (ci sono dati da scrivere)
 scrivi il dato sullo stream
chiudi lo stream
```

Come vedremo, la nozione di stream consente di definire i processi di lettura e di scrittura di informazione in modo indipendente dalla specifica sorgente o destinazione. Al momento della creazione di uno stream è necessario specificare qual è la sorgente o la destinazione dello stream, ma il processo di lettura/scrittura prescinde da questi dettagli.

All'interno del package `java.io` sono riconoscibili due sezioni principali: una per gestire stream di caratteri e una per gestire stream di byte. Nel primo caso, l'unità minima di informazione che viaggia lungo lo stream è costituita da un carattere, mentre nel secondo è costituita da un byte. Per orientarsi nella gerarchia delle classi che costituiscono il package `java.io` è bene ricordare che vengono utilizzati nomi differenti per le classi che gestiscono flussi di byte e flussi di caratteri. In particolare:

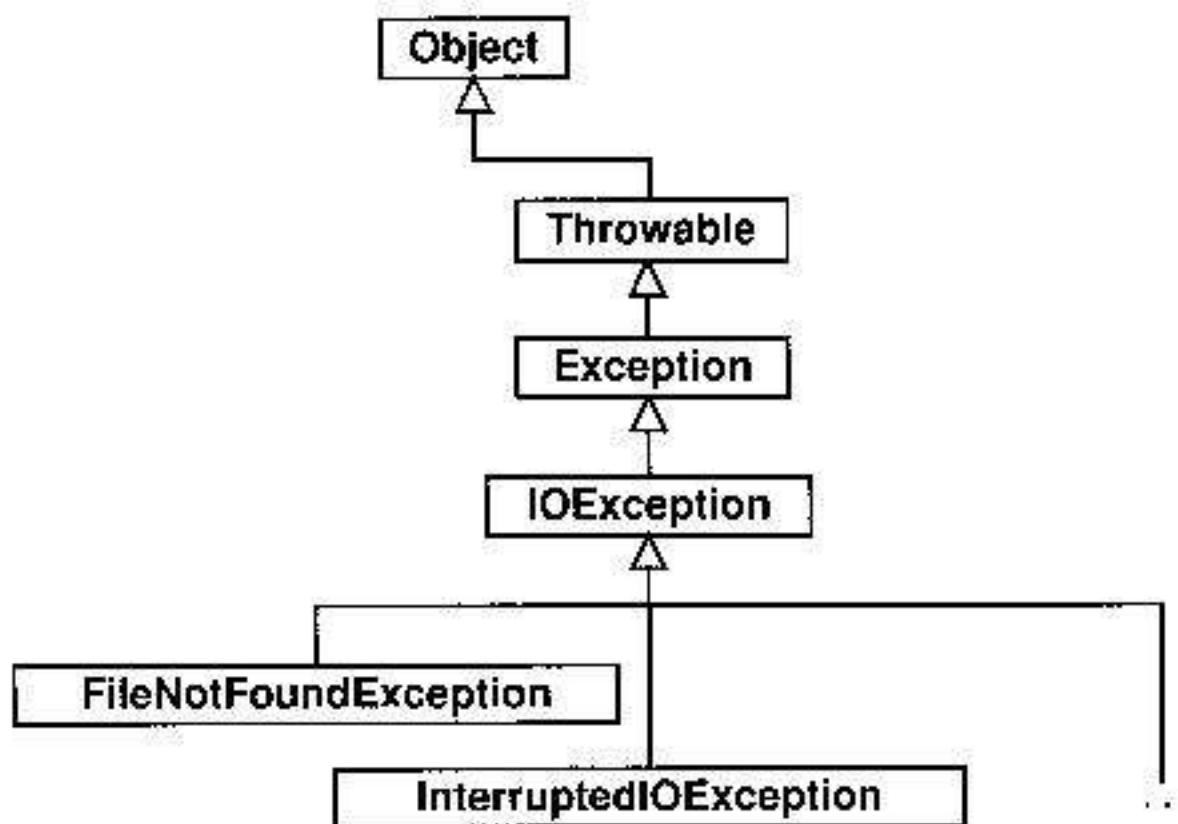
- le classi per la lettura di stream di byte contengono *InputStream* nel nome, mentre quelle per la scrittura di stream di byte contengono *OutputStream*;
- le classi per la lettura di stream di caratteri contengono *Reader* nel nome, mentre quelle per la scrittura di stream di caratteri contengono *Writer*.

Ad esempio la classe `FileInputStream` permette di leggere byte da uno stream collegato con un file, mentre la classe `FileReader` consente di leggere caratteri da uno stream collegato con un file.

Un aspetto importante delle classi per la gestione dell'input/output è che molti metodi di queste sollevano eccezioni controllate. Ogni volta che si tenta di leggere dati da una sorgente, o di scrivere dati su una destinazione, bisogna mettere in preventivo che le cose possano andare male. Ad esempio il collegamento potrebbe interrompersi mentre l'applicazione sta leggendo dati da un'altra macchina in rete; oppure mentre l'applicazione sta scrivendo dati in un file su disco, lo spazio potrebbe esaurirsi. Ovviamente queste anomalie non dipendono dal programma, e quindi ogni metodo che può risentirne deve segnalarlo mediante un'eccezione controllata. Le anomalie che possono verificarsi durante le operazioni di input/output sono modellate dalle eccezioni del package `java.io`. La classe `IOException` costituisce la radice della gerarchia di tali eccezioni; ne rappresentiamo un frammento nella Figura 13.1.

## 13.1 Stream di caratteri

Prima di iniziare a descrivere in dettaglio la gerarchia delle classi che compongono il package `prog.io`, proponiamo l'esempio di un'applicazione che legge un file di caratteri e lo visualizza. Per leggere un file di caratteri utilizzeremo la classe `FileReader`. Questa classe, le cui istanze rappresentano stream di caratteri in lettura collegati con una sorgente di tipo file, mette a disposizione, fra gli altri, il costruttore:



**Figura 13.1** Un frammento della gerarchia delle eccezioni di input/output del package `java.io`.

- `public FileReader(String nomeFile) throws FileNotFoundException`  
Crea un oggetto per leggere uno stream di caratteri collegato con il file il cui nome è specificato come argomento. Se il file con tale nome non esiste, il metodo solleva l'eccezione controllata `FileNotFoundException`.

Inoltre mette a disposizione, fra gli altri, i metodi:

- `public int read() throws IOException`  
Legge un singolo carattere dallo stream che esegue il metodo e ne restituisce il codice sotto forma di intero compreso tra 0 e 65535 (i numeri interi rappresentabili con 16 bit). Se si è raggiunta la fine del file, il metodo restituisce -1.
- `public int read(char[] buf) throws IOException`  
Legge dallo stream che esegue il metodo una sequenza di caratteri della stessa lunghezza dell'array specificato come argomento e li memorizza in esso.
- `public void close() throws IOException`  
Chiude lo stream che esegue il metodo. Dev'essere invocato per rilasciare la risorsa, cioè il file cui è collegato lo stream. Una volta che lo stream è stato chiuso, eventuali invocazioni di un metodo `read` (o di altri metodi che eseguono operazioni sullo stream) sollevano l'eccezione `IOException`.

Utilizzando la classe `FileReader` possiamo realizzare una semplice applicazione che visualizza il contenuto del file fornito come argomento al metodo `main`.

```

import prog.io.*;
import java.io.FileReader;

```

```

import java.io.IOException;

public class VisualizzaFile {

 public static void main(String[] args) throws IOException {
 //predisposizione del canale di output
 ConsoleOutputManager out = new ConsoleOutputManager();
 //costruzione dello stream di caratteri
 FileReader frd = new FileReader(args[0]);

 int i;
 //lettura e visualizzazione
 while ((i = frd.read()) != -1)
 out.print((char)i);

 //chiusura dello stream
 frd.close();
 }
}

```

Per prima cosa osserviamo che il metodo `main` delega la classe di eccezioni `IOException`. Ciò è dovuto al fatto che non trattiamo esplicitamente nel corpo del metodo `main` le eccezioni controllate `FileNotFoundException` e `IOException`, sollevate rispettivamente dal costruttore di `FileReader` e dal metodo `read`. Osserviamo inoltre che il codice per leggere un carattere dal file e visualizzarlo sul video è complicato dal fatto che il metodo `read` restituisce un `int` e non un `char`; quindi per visualizzare il carattere letto è necessario effettuare un cast dell'intero restituito dal metodo `read` al tipo `char`. È stata fatta questa scelta nel contratto del metodo `read` per disporre di un valore (`-1`, che non corrisponde al codice di alcun carattere) per comunicare a chi invoca il metodo che l'input è terminato.

È evidente che i metodi `read` forniti dalla classe non sono del tutto soddisfacenti; sarebbe più comodo ed elegante disporre di metodi che consentano di leggere dal file una linea alla volta anziché un carattere soltanto (o una sequenza di caratteri). Questa funzionalità è fornita da una classe diversa, la classe `BufferedReader`, che permette di leggere testo da uno stream di caratteri fornendo un meccanismo di bufferizzazione: un oggetto di questa classe preleva dallo stream un certo numero di caratteri alla volta e li conserva in una sua memoria interna (*buffer*). Da una parte questo consente di rendere più efficiente il processo di lettura dallo stream, dall'altra permette di organizzare le sequenze di caratteri provenienti dallo stream in strutture più complesse. Ad esempio la classe `BufferedReader` mette a disposizione il seguente metodo:

- `public String readLine() throws IOException`

Legge una linea di testo. Una linea è considerata conclusa dai caratteri `\n` (*linefeed*), `\r` (*carriage return*) o da un *carriage return* seguito da un *linefeed*. Se al momento dell'invocazione del metodo non si è raggiunta la fine dello stream, il metodo restituisce la stringa

contenente la linea di caratteri letta (senza il carattere di terminazione della riga). Se invece al momento dell'invocazione è stata raggiunta la fine dello stream, il metodo restituisce `null`.

La classe `BufferedReader` prescinde dalla specifica sorgente da cui provengono i dati. Infatti il costruttore della classe richiede come argomento un oggetto di tipo `Reader` che, come vedremo meglio in seguito, è la classe (astratta) estesa da tutte le classi per la lettura di stream di caratteri. Ad esempio, dato che la classe `FileReader` è una sottoclasse di `Reader`, ogni riferimento di tipo `FileReader` può essere utilizzato come argomento del costruttore di `BufferedReader`. Possiamo quindi riscrivere l'applicazione precedente così:

```
import prog.io.*;
import java.io.FileReader;
import java.io.IOException;
import java.io.BufferedReader;

public class VisualizzaFile {

 public static void main(String[] args) throws IOException {
 //predisposizione del canale di output
 ConsoleOutputManager out = new ConsoleOutputManager();
 //costruzione dello stream di caratteri
 FileReader frd = new FileReader(args[0]);
 BufferedReader bfr = new BufferedReader(frd);

 String str;
 //lettura e visualizzazione
 while ((str = bfr.readLine()) != null)
 out.println(str);

 //chiusura dello stream
 bfr.close();
 frd.close();
 }
}
```

Si osservi che per chiarezza abbiamo costruito prima un oggetto di tipo `FileReader`, di cui abbiamo memorizzato il riferimento nella variabile `frd`, e quindi abbiamo utilizzato questa variabile come argomento nel costruttore di `BufferedReader`. Più semplicemente avremmo potuto utilizzare l'unica istruzione:

```
BufferedReader bfr = new BufferedReader(new FileReader(args[0]));
```

Esiste una ragione ben precisa per la quale i progettisti del package `java.io` hanno distribuito i compiti fra le classi `BufferedReader` e `FileReader`. Quest’organizzazione consente di avere una classe di riferimento (`BufferedReader`) per la lettura delle linee di testo, indipendente dalla sorgente specifica da cui provengono i caratteri. Tutte le classi per la lettura del package `prog.io` (sia quelle per la lettura di stream di caratteri sia quelle per la lettura di stream di byte) sono organizzate in questo modo: alcune classi realizzano un collegamento con una specifica sorgente (come `FileReader`) con semplici metodi di lettura (`read` di caratteri o byte), altre permettono di costruire dati strutturati a partire da tali collegamenti. Un discorso analogo vale anche per le classi che scrivono stream.

Quest’organizzazione risulta molto utile quando si devono realizzare classi per leggere/scrivere da una sorgente/destinazione non trattata dal package `java.io`; infatti è sufficiente realizzare i metodi per leggere/scrivere caratteri o byte e utilizzare poi le classi già disponibili nel package per leggere/scrivere dati più complessi.

## 13.2 Le gerarchie Reader e Writer

Per modellare la lettura e la scrittura di stream di caratteri, il package `java.io` prevede due classi astratte, denominate `Reader` e `Writer`. Queste classi sono astratte in quanto il loro scopo è quello di definire l’insieme dei metodi di base per la gestione di uno stream di caratteri a prescindere dallo stream specifico. L’implementazione dei metodi per la lettura e la scrittura dei caratteri è demandata a sottoclassi concrete, che sono invece realizzate tenendo conto di sorgenti/destinazioni specifiche.

I metodi definiti dalla classe `Reader` sono essenzialmente quelli che abbiamo descritto in precedenza per la classe `FileReader`. Nella Figura 13.2 è illustrata una parte della gerarchia sottostante a `Reader`. Le classi evidenziate in grigio sono le classi che leggono direttamente caratteri da una sorgente (*data sink* nella terminologia di Java), le altre forniscono invece meccanismi per manipolare le informazioni lette.

Osserviamo che anche la classe `BufferedReader` è un’istanza di `Reader`, e di conseguenza fornisce un’implementazione dei metodi della classe `Reader`. Si tratta in realtà di una classe utilizzata comunemente come `Reader` per questioni di efficienza anche quando non è necessario organizzare i caratteri in linee. Infatti un oggetto `Reader` deve di solito prelevare un carattere dalla sorgente di dati ogni volta che esegue il metodo `read()`. Se i tempi di accesso alla risorsa su cui risiede la sorgente sono molto lunghi, come quando la sorgente è un file, è opportuno leggere molti caratteri in una sola volta e memorizzarli in un buffer, in modo che la prossima lettura prelevi il carattere dal buffer anziché richiedere un nuovo accesso alla risorsa.

Per quel che riguarda le sottoclassi che leggono direttamente da una sorgente (quelle in grigio nella Figura 13.2), osserviamo che, a parte la già menzionata `FileReader`, che consente di leggere caratteri da un file, e la classe `PipedReader`, che non tratteremo, ci sono due classi, `CharArrayReader` e `StringReader`, che permettono di leggere caratteri da sorgenti di dati in memoria. La seguente applicazione fornisce un esempio di lettura da stream collegati a una stringa e a un array di `char` in memoria.

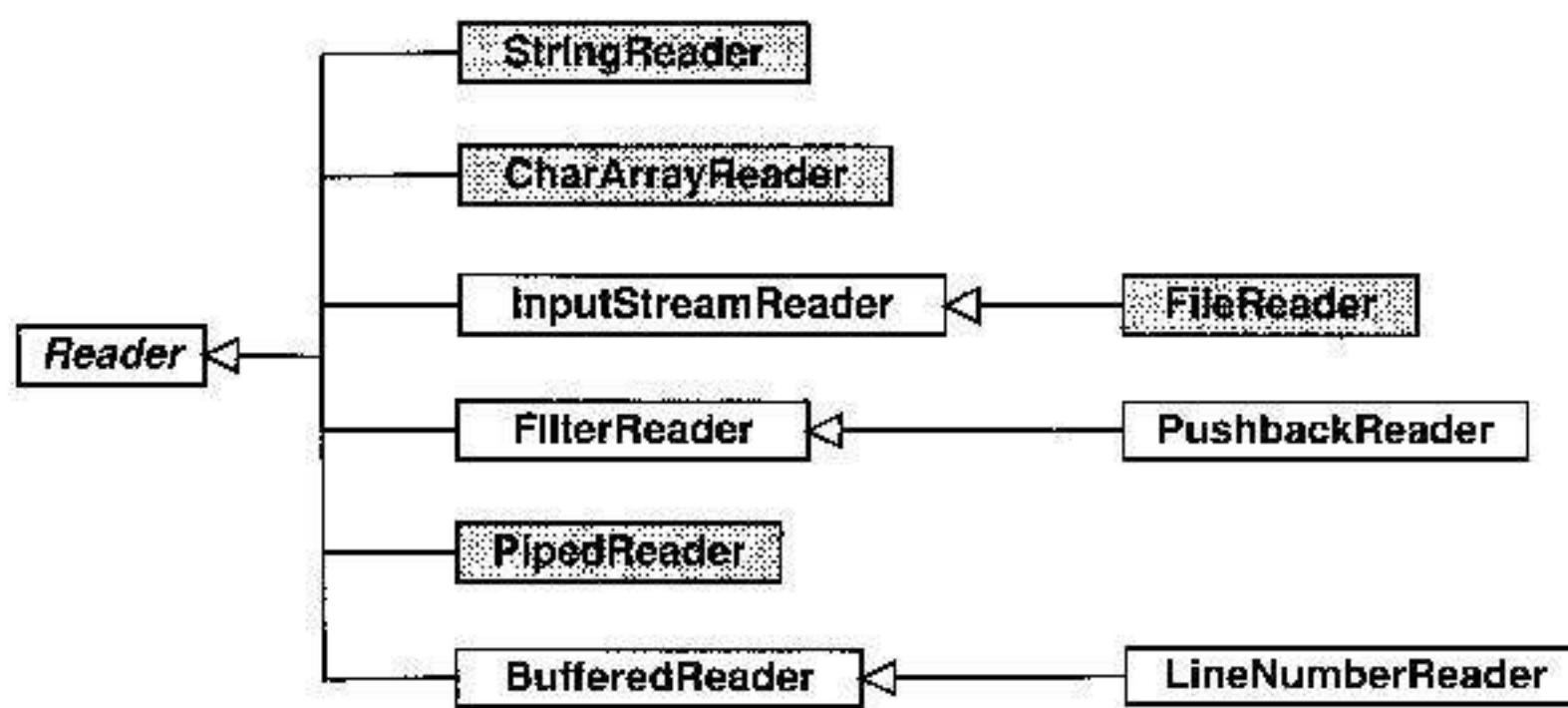


Figura 13.2 La gerarchia di Reader.

```

import prog.io.*;
import java.io.*;

public class LetturaDaMemoria {

 public static void main(String[] args) throws IOException {
 //predisposizione del canale di output
 ConsoleOutputManager out = new ConsoleOutputManager();

 int i; //per memorizzare il codice del carattere letto

 //stream collegato a una stringa in memoria
 StringReader sr = new StringReader("pippo");
 //lettura e visualizzazione
 while ((i = sr.read()) != -1)
 out.print((char)i);
 sr.close();

 //stream collegato a un array di char in memoria
 char[] a = {'a','b','c'};
 CharArrayReader cr = new CharArrayReader(a);

 //lettura e visualizzazione
 while ((i = cr.read()) != -1)
 out.print((char)i);
 cr.close();
 }
}

```

}

Osserviamo infine che la classe `FileReader` è una sottoclasse della classe `InputStreamReader`. Gli oggetti di questa classe permettono di tradurre uno stream di byte in uno stream di caratteri. In effetti un `FileReader` è un traduttore particolare che converte in caratteri i byte che costituiscono il file.

La classe astratta `Writer` è l'analogo per la scrittura di stream di caratteri della classe `Reader`. I metodi definiti in questa classe sono metodi per la gestione dello stream (come il metodo `close` per chiudere uno stream) e semplici metodi `write` per scrivere un singolo carattere, array di caratteri, o una stringa. Fra i metodi per la gestione dello stream segnaliamo in particolare il seguente:

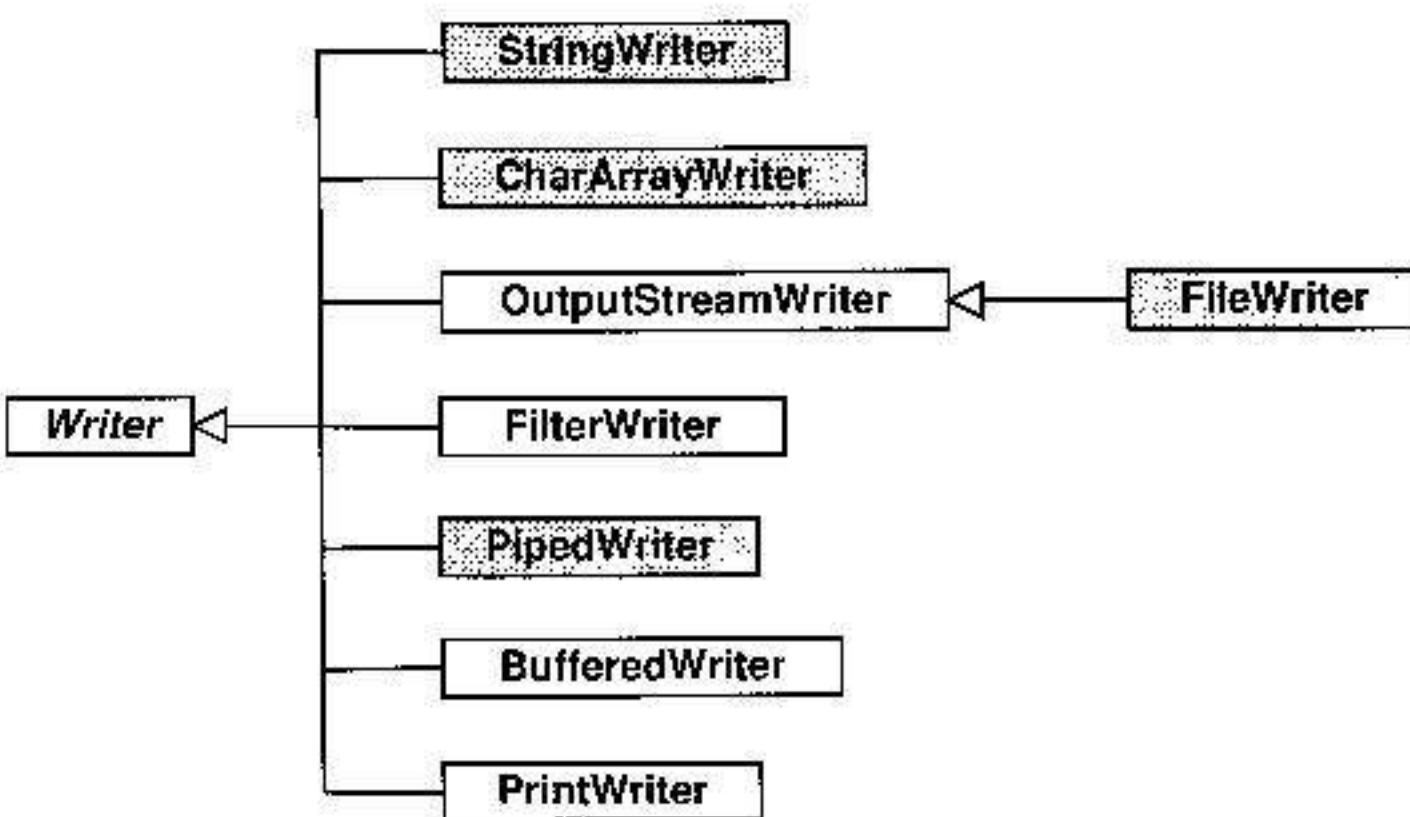
- `public abstract void flush() throws IOException`  
Forza la scrittura dei caratteri sullo stream che esegue il metodo.

Per comprendere il significato del metodo `flush` bisogna fare alcune premesse. Esistono casi in cui l'operazione di scrittura su uno stream può essere molto inefficiente in termini di tempo. Ad esempio scrivere su un file su disco richiede tempi lunghi perché l'accesso al supporto magnetico è mediato da componenti meccaniche. Per questa ragione è preferibile scrivere un blocco di caratteri anziché un singolo carattere alla volta. In casi come questo, l'effetto di una `write` non è quindi quello di scrivere effettivamente sullo stream, ma di scrivere i caratteri in una memoria temporanea (buffer) dove vengono conservati fino a quando non è disponibile una quantità sufficiente di caratteri. Invocando il metodo `flush` si forza il sistema a trasferire effettivamente sullo stream tutti i dati attualmente presenti nel buffer.

Nella Figura 13.3 è illustrata una parte della gerarchia sottostante a `Writer`. Le classi evidenziate in grigio sono le classi che scrivono direttamente su una destinazione, le altre forniscono invece un qualche meccanismo per manipolare i dati. Si osservi l'analogia fra la gerarchia di `Writer` e quella di `Reader`. Come c'è una classe per leggere da file, così troviamo una classe `FileWriter` per scrivere su file di caratteri; come disponiamo di classi per leggere caratteri da un array e da una stringa, così esistono le analoghe classi per la scrittura `CharArrayWriter` e `StringWriter`. La classe che gestisce lo stream collegato al file è una sottoclasse di `OutputStreamWriter`. Le istanze di questa classe realizzano la traduzione di uno stream di caratteri in uno stream di byte. In effetti, per scrivere caratteri su un file, la classe `FileWriter` deve effettuare la traduzione di un carattere nella sequenza di byte che lo rappresenta.

Come esempio proponiamo un'applicazione per copiare file di caratteri. Per copiare il file dovremo aprirlo e leggere un carattere alla volta utilizzando uno stream `FileReader` in modo analogo a quanto fatto nella precedente applicazione `VisualizzaFile.java`. Per scrivere il file di caratteri utilizzeremo invece la classe `FileWriter`. Questa classe mette a disposizione, fra gli altri, i seguenti costruttori:

- `public FileWriter(String fileName) throws IOException`  
Crea un oggetto per scrivere su uno stream di caratteri collegato al file il cui nome è specificato come argomento. Se il file con il nome specificato non esiste, allora viene creato;



**Figura 13.3** La gerarchia di Writer.

se invece esiste, il suo contenuto viene sovrascritto. Se il nome fornito come argomento corrisponde a un file che non può essere scritto, oppure a una directory, viene sollevata un'eccezione di tipo IOException.

- `public FileWriter(String fileName, boolean append) throws IOException`  
Crea un oggetto `FileWriter` per scrivere sul file di caratteri il cui nome è specificato come argomento. Se il file con il nome specificato non esiste, allora viene creato. Se il file esiste e il valore di `append` è `false`, allora il file viene sovrascritto, mentre se il valore di `append` è `true`, allora i caratteri scritti sullo stream verranno accodati all'attuale contenuto del file. Viene sollevata un'eccezione di tipo `IOException` se il nome fornito come argomento corrisponde a un file che non può essere scritto, oppure a una directory.

I seguenti metodi della classe `FileWriter` sono tutti già previsti dalla classe astratta `Writer`.

- `public void write(int c) throws IOException`  
Scrive `c` sotto forma di carattere sullo stream di caratteri che esegue il metodo. Il carattere viene passato come argomento di tipo `int`. Tuttavia sono presi in considerazione solo i 16 bit meno significativi dell'`int`.
- `public void write(char[] buf, int offset, int c) throws IOException`  
Scrive sullo stream che esegue il metodo i `c` caratteri dell'array `buf` contenuti a partire dalla posizione `offset`, cioè i caratteri `buf[offset]`, `buf[offset + 1]`, ..., `buf[offset + c - 1]`.
- `public void write(char[] buf) throws IOException`  
Scrive i caratteri dell'array `buf` specificato come argomento sullo stream che esegue il metodo. Corrisponde a `write(buf, 0, buf.length)`.

- **public void write(String s, int offset, int c) throws IOException**  
Scrive sullo stream che esegue il metodo c caratteri della stringa s a partire dalla posizione s.charAt(offset).
- **public void write(String s) throws IOException**  
Scrive i caratteri della stringa s fornita come argomento sullo stream che esegue il metodo. Corrisponde a write(s, 0, s.length).
- **public void close() throws IOException**  
Chiude lo stream rilasciando la risorsa, cioè il file cui è collegato lo stream. Una volta che lo stream è stato chiuso, eventuali invocazioni di un metodo read (o di altri metodi che eseguano operazioni sullo stream) sollevano l'eccezione IOException.
- **public void flush() throws IOException**  
Forza la scrittura dei caratteri sullo stream.

Possiamo presentare direttamente la semplice applicazione per copiare un file. Il nome del file da copiare e il nome del nuovo file devono essere specificati dalla linea di comando.

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

public class CopiaFile {

 public static void main(String[] args) throws IOException {
 if (args.length == 2) {
 //stream per la lettura del file
 FileReader frd = new FileReader(args[0]);
 //stream per la scrittura del file
 FileWriter fwr = new FileWriter(args[1]);

 int i;
 //lettura e visualizzazione
 while ((i = frd.read()) != -1)
 fwr.write(i);

 //flush e chiusura dello stream per la scrittura
 fwr.flush();
 fwr.close();

 //chiusura dello stream per la lettura
 frd.close();
 }
 }
}
```

```

 ConsoleOutputManager out = new ConsoleOutputManager();
 out.println("Uso: java CopiaFile <file_da_copiare> <file_copia>");
}
}
}

```

La classe `BufferedWriter` mette a disposizione gli stessi metodi descritti dalla classe astratta `Writer`, ma fornisce un meccanismo di bufferizzazione che rende più efficienti le operazioni di scrittura. Mette a disposizione due costruttori: il primo che riceve semplicemente un argomento di tipo `Writer` e il secondo che riceve anche un secondo argomento di tipo `int`, che indica la dimensione del buffer. Ad esempio, se nell'applicazione precedente sostituiamo l'istruzione

```
FileWriter fwr = new FileWriter(args[1]);
```

con l'istruzione

```
BufferedWriter fwr = new BufferedWriter(new FileWriter(args[1]));
```

otteniamo un'applicazione equivalente dal punto di vista dell'effetto, che dovrebbe però risultare più efficiente nell'accesso al file in fase di scrittura.

Una classe molto utile per scrivere su stream di caratteri è la classe `PrintWriter`. Questa classe permette di scrivere su uno stream tipi primitivi e oggetti in forma testuale. La classe mette a disposizione, fra gli altri, il seguente costruttore:

- `public PrintWriter(Writer out)`

Crea un oggetto `PrintWriter` per scrivere sullo stream di caratteri `Writer` specificato come argomento.

La classe mette inoltre a disposizione metodi per scrivere in formato carattere tutti i tipi primitivi. Ad esempio il metodo `print(int i)` scrive sullo stream la sequenza di caratteri corrispondente alla stringa `String.valueOf(i)`. Per ogni metodo `print` la classe fornisce anche il corrispondente metodo `println` che, dopo aver scritto il valore fornito come argomento, scrive sullo stream un separatore di linea. Ad esempio quest'applicazione scrive sul file di nome "prova.txt" alcuni valori di tipo primitivo.

```

import java.io.*;

public class EsempioPrintWriter {

 public static void main(String[] args) throws IOException {
 //PrintStream collegato con il file
 PrintWriter pwr = new PrintWriter(new FileWriter("prova.txt"));

 pwr.println(1234);
 pwr.println("pippo");
 }
}

```

```

 pwr.println(3.14);
 pwr.println(true);

 pwr.flush();
 pwr.close();
}
}

```

## Esercizi

- 13.1 Riscrivete l'applicazione `VisualizzaFile` in modo che visualizzi il numero di linea all'inizio di ogni linea che compone il file. Scrivete l'applicazione in modo che specifichi dalla linea di comando se deve visualizzare il numero di riga o no.
- 13.2 Riscrivete l'applicazione `CopiaFile` copiando il file riga per riga anziché carattere per carattere (utilizzate le classi `BufferedReader` e `BufferedWriter`).

### 13.3 La classe `File`

La classe `File` è una classe di utilità i cui oggetti forniscono una rappresentazione astratta dei file e mettono a disposizione diversi metodi utili per la loro gestione. Tutte le classi che realizzano stream per leggere o scrivere file, sia che si tratti di stream di caratteri sia che si tratti di stream di byte, offrono un costruttore che riceve come argomento un oggetto di tipo `File`. Ad esempio la classe `FileReader` descritta nel paragrafo precedente fornisce anche il seguente costruttore:

- `public FileReader(File file) throws FileNotFoundException`  
Crea un oggetto per leggere stream di caratteri collegato con il file rappresentato dall'oggetto di tipo `File` specificato come argomento. Se il file rappresentato da tale oggetto non esiste, il metodo solleva l'eccezione controllata `FileNotFoundException`.

La caratteristica principale della classe `File` è di fornire una rappresentazione dei nomi dei file e delle directory (*pathname*, percorsi) indipendente dal sistema operativo. Le informazioni relative alla rappresentazione dei pathname nel sistema operativo corrente sono disponibili mediante alcuni campi statici della classe. Ad esempio:

- `public static final char separator`  
Contiene il carattere utilizzato dal sistema operativo per separare le componenti del pathname. In un sistema UNIX o Linux il valore di questo campo è `/`.

La classe mette a disposizione, fra gli altri, il seguente costruttore:

- `public File(String pathname)`  
Crea un oggetto di tipo `File` che permette di manipolare il `pathname` specificato.

Inoltre la classe mette a disposizione, fra gli altri, metodi per ottenere informazioni sul file fisico rappresentato dall'oggetto di tipo `File`. Alcuni di questi sono:

- `public boolean exists()`  
Verifica se il file rappresentato dall'oggetto che esegue il metodo esiste. Restituisce `true` in caso affermativo.
- `public boolean isDirectory()`  
Verifica se il file associato all'oggetto che esegue il metodo esiste ed è una *directory*, nel qual caso il metodo restituisce `true`.
- `public boolean isFile()`  
Verifica se il file associato all'oggetto che esegue il metodo è un file *normale*, nel qual caso il metodo restituisce `true`. Un file è *normale* se non è una *directory* e soddisfa altre proprietà che dipendono dal sistema operativo.
- `public String getName()`  
Restituisce il nome del file (o della directory) identificata dall'oggetto che esegue il metodo. Il nome restituito è l'ultima componente del pathname completo del file.
- `public String getAbsolutePath()`  
Restituisce la stringa che descrive il pathname completo del file.
- `public long lastModified()`  
Restituisce un `long` che rappresenta la data dell'ultima modifica del file identificato dall'oggetto che esegue il metodo. La data dell'ultima modifica è espressa come numero di millisecondi trascorsi dalle ore 00:00:00 GMT del primo gennaio 1970.<sup>1</sup>
- `public long length()`  
Restituisce la dimensione, cioè il numero di byte, del file identificato dall'oggetto che esegue il metodo.
- `public boolean canWrite()`  
Controlla se il file può essere scritto. Restituisce `true` se e solo se il file identificato dall'oggetto che esegue il metodo può essere scritto.
- `public boolean canRead()`  
Controlla se il file può essere letto. Restituisce `true` se e solo se il file identificato dall'oggetto che esegue il metodo può essere letto.
- `String[] list()`  
Restituisce un array di stringhe contenenti i nomi dei file e delle directory presenti nella directory rappresentata dall'oggetto di tipo `File` che esegue il metodo. Restituisce `null` se l'oggetto che esegue il metodo non rappresenta una directory o se si verifica un errore di input/output.

<sup>1</sup> Le date espresse in questo formato possono essere tradotte in date nel formato comune tramite l'apposito costruttore con argomento `long` della classe `Date` del package `java.util` (si veda l'applicazione `TestFile.java`).

La seguente applicazione fornisce informazioni sul file o la directory specificata come argomento sulla riga di comando.

```
import prog.io.*;
import java.io.*;
import java.util.Date;

public class TestFile {

 public static void main(String[] args) throws IOException {
 ConsoleOutputManager out = new ConsoleOutputManager();

 //le costanti
 out.println("Separatore utilizzato dal sistema operativo = "
 + File.separator);

 File file = new File(args[0]);
 // informazioni sul file
 out.println(" nome = " + file.getName());
 out.println(" pathname assoluto = " + file.getAbsolutePath());
 out.println(" directory padre = " +
 (new File(file.getAbsolutePath()).getParent()));
 out.println(" esiste? = " + file.exists());
 out.println(" è leggibile? = " + file.canRead());
 out.println(" è scrivibile? = " + file.canWrite());
 out.println(" è un file? = " + file.isFile());
 out.println(" è una directory? = " + file.isDirectory());
 out.println(" ultima modifica = " +
 (new Date(file.lastModified())).toString());
 out.println(" dimensione = " + file.length() + " byte");
 }
}
```

Quelli che seguono sono due esempi di esecuzione sul sistema in cui abbiamo sviluppato l'applicazione: la prima fornendo come argomento il nome del file `TestFile.java` e la seconda fornendo come argomento il nome della directory corrente (cioè `.`).

```
> java TestFile TestFile.java
Separatore utilizzato dal sistema operativo =
 nome = TestFile.java
pathname assoluto = /home/ferram/javaBook/c12-stream/java/TestFile.java
 directory padre = /home/ferram/javaBook/c12-stream/java
 esiste? = true
```

```

 è leggibile? = true
 è scrivibile? = true
 è un file? = true
è una directory? = false
ultima modifica = Sun Mar 23 18:25:12 CET 2003
dimensione = 1044 byte

> java TestFile .
Separatore utilizzato dal sistema operativo = /
 nome =
pathname assoluto = /home/ferram/javaBook/c12-stream/java/ .
 directory padre = /home/ferram/javaBook/c12-stream/java
 esiste? = true
 è leggibile? = true
 è scrivibile? = true
 è un file? = false
è una directory? = true
ultima modifica = Sun Mar 23 18:25:12 CET 2003
dimensione = 4096 byte

```

Si osservi che, nell'esecuzione precedente, il metodo `exists` restituisce `true` perché il file identificato dall'oggetto cui si riferisce `file` esiste. Se tale file non esistesse, l'oggetto che lo descrive verrebbe ugualmente creato, ma in questo caso il metodo `exists` restituirebbe `false` e gli altri metodi che restituiscono informazioni su un file si comporterebbero in modo consistente. Quindi creare un oggetto di tipo `File` non significa creare un file. È possibile forzare la creazione del file identificato da un oggetto di tipo `File` mediante il metodo `createNewFile`; lo descriviamo qui di seguito insieme ad alcuni metodi per operare su file e directory.

- **public boolean createNewFile() throws IOException**  
Crea un nuovo file vuoto corrispondente all'oggetto che esegue il metodo. Il file viene creato se e solo se non esiste già un file con il medesimo nome. Restituisce `true` se il file non esiste e viene creato con successo, restituisce `false` se il file esiste già.
- **public boolean mkdir()**  
Crea una directory corrispondente all'oggetto che esegue il metodo. Restituisce `true` se e solo se la directory viene creata con successo.
- **public boolean delete()**  
Cancella il file o la directory relativa all'oggetto che esegue il metodo. Se il pathname denota una directory, allora viene cancellata solo a patto che sia vuota. Restituisce `true` se e solo se l'operazione ha successo.
- **public void deleteOnExit()**  
Richiede che il file o la directory cui fa riferimento l'oggetto che esegue il metodo sia

cancellata dalla macchina virtuale prima della sua terminazione. Una volta richiesta, la cancellazione non può più essere sospesa.

La classe `File` mette a disposizione anche due metodi statici, denominati `createTempFile`, per la creazione di file temporanei. I file temporanei sono utili quando si sviluppano applicazioni che devono manipolare grandi quantità di dati che non possono essere conservate in memoria. I metodi `createTempFile` creano file in modo da garantire che il loro nome non entri in conflitto con quello di file già esistenti. Per una descrizione dettagliata di questi metodi e degli altri metodi forniti dalla classe `File` si rimanda alla documentazione.

Per concludere questo paragrafo presentiamo, senza ulteriori commenti, un'applicazione che stampa la lista dei file e delle directory presenti nella directory fornita come argomento.

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

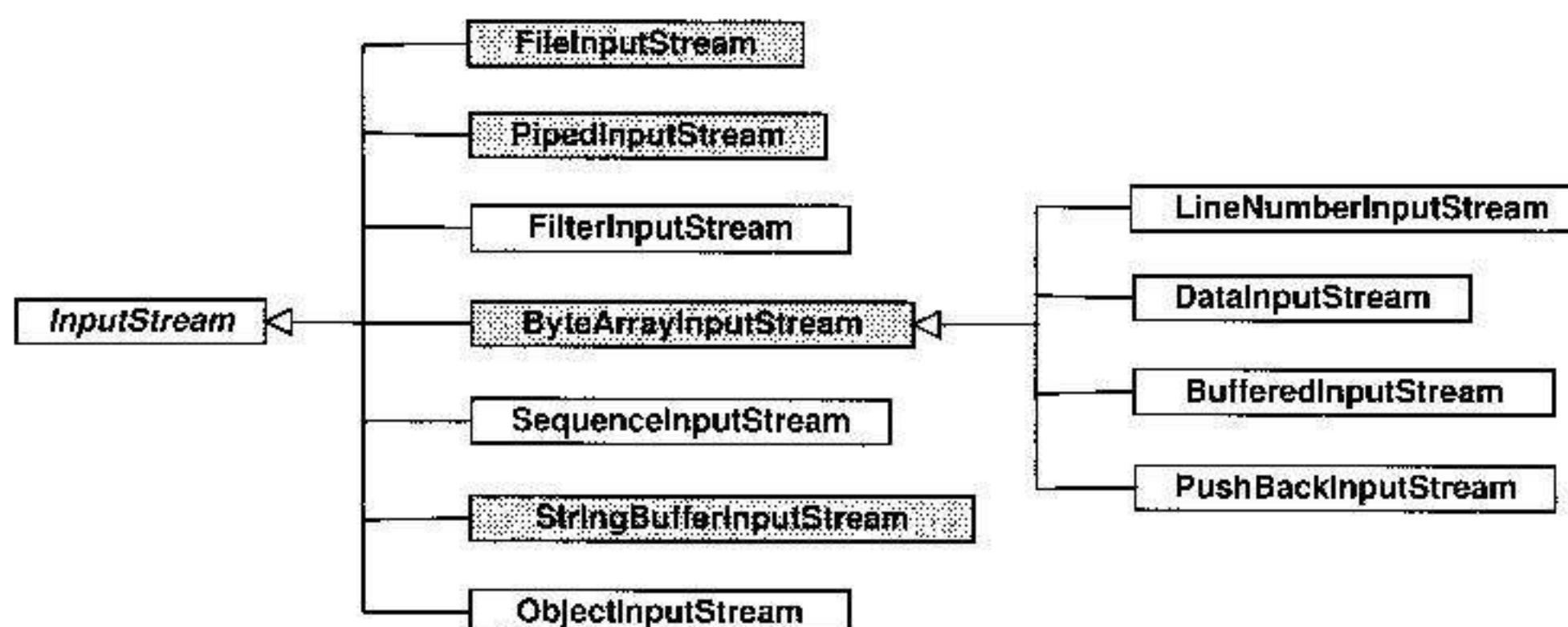
public class ListaDir {

 public static void main(String[] args) throws IOException {
 //predisposizione del canale di output
 ConsoleOutputManager out = new ConsoleOutputManager();

 File f = null;
 //se non ci sono argomenti consideriamo la directory corrente
 if (args.length == 0)
 f = new File(".");
 else
 f = new File(args[0]);

 //se il nome specificato è...
 if (f.isFile())
 //...quello di un file allora ne stampa nome e dimensione
 out.println("File: " + f.getAbsolutePath() + ", " + f.length()
 + " byte");
 else {
 //...quello di una dir allora...
 out.println("Directory: " + f.getAbsolutePath());
 //...preleva la lista dei file...
 String[] lista = f.list();
 //...stampa i dati di ognuno
 for(int i=0; i < lista.length; i++){
 File tmp = new File(f.getAbsolutePath(), lista[i]);
 if (tmp.isFile())
 out.println(" file: " + tmp.getName() + " " + tmp.length())
 }
 }
 }
 }
```

```
 + " byte");
 else
 out.println(" dir. : " + tmp.getName());
 }
}
}
```



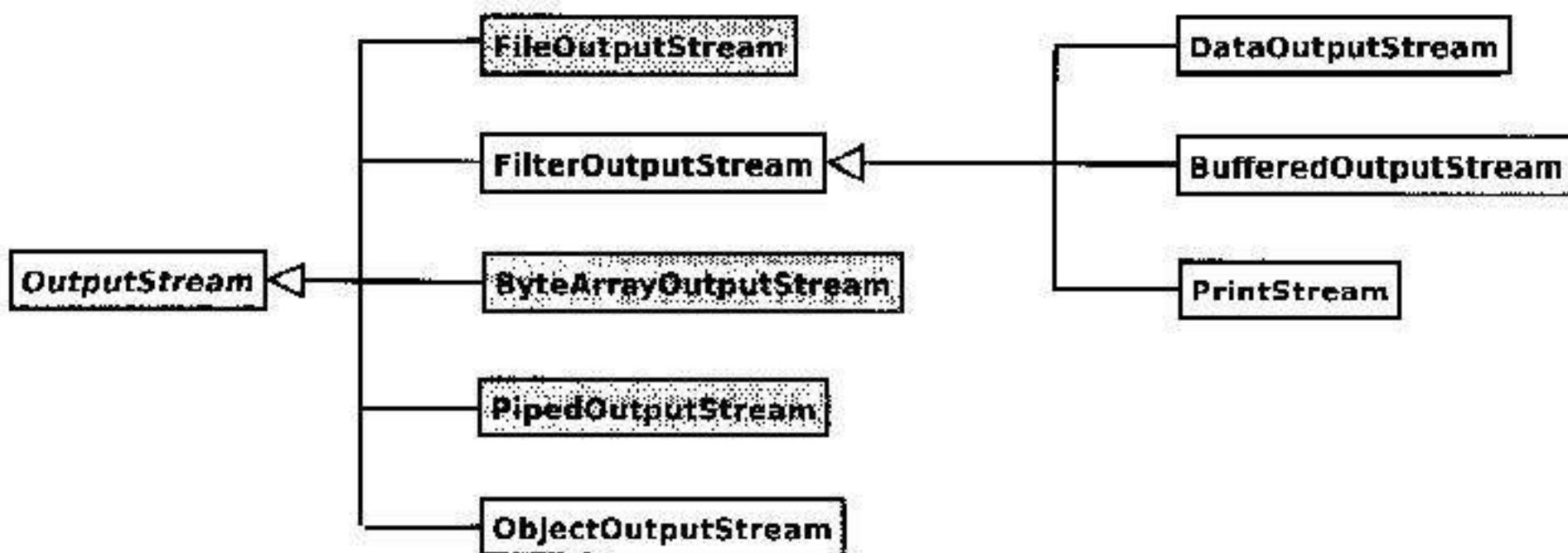
**Figura 13.4** La gerarchia di `InputStream`.

### 13.4 Stream di byte

Le gerarchie delle classi per leggere e scrivere stream di byte sono parallele a quelle per leggere e scrivere stream di caratteri. All’apice delle corrispondenti gerarchie troviamo le classi astratte `InputStream` e `OutputStream`, che definiscono i metodi per gestire lo stream e semplici operazioni di lettura e scrittura di byte. I metodi definiti da queste classi sono analoghi a quelli delle classi `Reader` e `Writer`.

La classe astratta `InputStream` fornisce i metodi per la lettura di un singolo byte (`read()`<sup>2</sup>), di array di byte (`read(byte[] b)`) e per la gestione dello stream: `close()` per chiudere lo stream e `available()` che restituisce il numero di byte che possono essere letti dallo stream. La parte della gerarchia delle classi di `java.io` sottostante a `InputStream` è illustrata nella Figura 13.4. Anche in questo caso le classi evidenziate in grigio sono quelle che permettono di creare stream di byte collegati a una sorgente, mentre le altre forniscono un qualche tipo di elaborazione dei dati.

<sup>2</sup> Il metodo `read()` restituisce un `int` come l'analogo metodo della classe `Reader`. Ciò permette di disporre di un valore (`-1`) al di fuori dell'insieme dei valori rappresentati da un `byte`, tramite il quale segnalare la terminazione dello stream.



**Figura 13.5** La gerarchia di `OutputStream`.

Analogamente la classe `OutputStream` definisce semplici metodi per la scrittura di un byte (`write(int b)`<sup>3</sup>) e di un array di byte (`write(byte[] b)`) e metodi per la gestione dello stream come `close()` e `flush()`. La gerarchia delle classi con al vertice `OutputStream` è rappresentata nella Figura 13.5.

## 13.5 Lettura e scrittura di dati primitivi

In questo paragrafo consideriamo il problema di scrivere e leggere valori di tipo primitivo su uno stream di byte. Le funzionalità necessarie sono fornite dalle classi `DataOutputStream` e `DataInputStream`. Ognuna di queste classi mette a disposizione un solo costruttore.

- `DataOutputStream(OutputStream out)`  
Costruisce un'istanza dello stream per scrivere sul sottostante stream di byte.
- `DataInputStream(InputStream in)`  
Costruisce un'istanza dello stream per leggere dal sottostante stream di byte.

Le classi mettono a disposizione, fra gli altri, metodi di scrittura e di lettura per ogni tipo primitivo. I prototipi dei metodi sono riassunti in questa tabella:

<sup>3</sup> L'argomento viene fornito come `int`, ma solo gli otto bit meno significativi dell'intero vengono scritti sullo stream. Questa scelta è stata fatta per evitare di dover effettuare un cast al tipo `byte` ogni volta che si intende stampare il risultato di un'espressione aritmetica con operandi di tipo `byte`. Si ricordi che tutti gli operatori applicabili a operandi di tipo `byte`, come ad esempio `+` e `*`, restituiscono un risultato di tipo `int`.

| DataInputStream       | DataOutputStream             |
|-----------------------|------------------------------|
| boolean readBoolean() | void writeBoolean(boolean v) |
| byte readByte()       | void writeByte(int v)        |
| char readChar()       | void writeChar(int v)        |
| short readShort()     | void writeShort(int v)       |
| int readInt()         | void writeInt(int v)         |
| long readLong()       | void writeLong(long v)       |
| float readFloat()     | void writeFloat(float v)     |
| double readDouble()   | void writeDouble(double v)   |

Ad esempio il seguente metodo scrive sul file specificato come primo argomento i valori double memorizzati nell'array fornito come secondo argomento.

```
public void scriviDati(File f, double[] dati) throws IOException {
 DataOutputStream dos = new DataOutputStream(new FileOutputStream(f));

 for(int i = 0; i < dati.length; i++)
 dos.writeDouble(dati[i]);

 dos.close();
}
```

La lettura di valori di tipo double contenuti in un file è illustrata nel metodo seguente:

```
public double[] leggiDati(File f, int numeroDati) throws IOException {
 DataInputStream dis = new DataInputStream(new FileInputStream(f));
 double[] dati = new double[numeroDati];

 for(int i = 0; i < numeroDati; i++)
 dati[i] = dis.readDouble();

 dis.close();
 return dati;
}
```

Si osservi che in questo metodo abbiamo fornito come argomento il numero di valori da leggere. Utilizzando il metodo `available` è possibile riscrivere il metodo `leggiDati` per leggere un numero arbitrario di valori double da un file. Nel prossimo esempio utilizziamo una `LinkedList` per memorizzare, sotto forma di oggetti di tipo `Double`, i valori letti dal file:

```
public LinkedList leggiDati(File f) throws IOException {
 DataInputStream dis = new DataInputStream(new FileInputStream(f));
 LinkedList dati = new LinkedList();

 while (dis.available() >= 8)
```

```

 dati.add(new Double(dis.readDouble()));

 dis.close();
 return dati;
}

```

Non insistiamo oltre sull'utilizzo delle classi `DataInputStream` e `DataOutputStream`: come vedremo, la lettura e la scrittura di dati di tipo primitivo può essere semplificata usando le classi che realizzano flussi di oggetti. Aggiungiamo solo che queste classi implementano due interfacce denominate `DataInput` e `DataOutput`, che definiscono i metodi per la lettura e la scrittura di tipi primitivi. Queste interfacce sono implementate anche dalla classe `RandomAccessFile`, che permette di trattare i file come se fossero flussi di byte o di caratteri: è possibile accedere alle loro informazioni in modo diretto (come in un array) anziché in ordine sequenziale.

## 13.6 La classe PrintStream

Per concludere l'analisi dei flussi di byte presentiamo la classe per l'output di byte, `PrintStream`, che è importante in relazione alla scrittura sullo standard output (cioè il monitor). La classe `PrintStream` aggiunge funzionalità a uno stream di output. In particolare fornisce metodi per scrivere rappresentazioni di tipi primitivi e oggetti. Inoltre, diversamente da quanto accade per altri stream di output, i metodi `print` di questa classe non sollevano eccezioni controllate; nel codice di un'applicazione ciò consente di utilizzarli in modo più agevole. Ovviamente anche i metodi `print` di questa classe possono causare errori dovuti a malfunzionamenti della periferica cui è collegato lo stream; anziché sollevare un'eccezione, in caso di errore viene posta a `true` una variabile privata dell'oggetto, che può essere controllata dall'utente mediante il metodo `public boolean checkError()`: se si è verificata una `IOException` durante l'ultima operazione sullo stream, esso restituisce infatti `true`.

La classe mette a disposizione un metodo `print` e un metodo `println` per ogni tipo primitivo, per oggetti di tipo `String` e per oggetti di tipo `Object` (in quest'ultimo caso il metodo non fa altro che stampare la stringa restituita dal metodo `toString` dell'oggetto).

## 13.7 I flussi di input/output standard

A questo punto possiamo chiarire in che modo avviene la lettura da standard input (la tastiera) e la scrittura sullo standard output e lo standard error (il video).

Per quel che riguarda standard output e standard error, si tratta di due stream di tipo `PrintStream`. I riferimenti a questi stream sono conservati in due campi statici della classe `System` del package `java.lang`. Per la precisione:

- `System.out` è di tipo `PrintStream` e rappresenta lo standard output comunemente utilizzato per l'output dei programmi: tipicamente (ma non obbligatoriamente) viene mostrato a video.

- `System.err` è di tipo `PrintStream` e rappresenta lo standard error, cioè lo stream cui dovrebbero essere inviati i messaggi di errore.

Per quel che riguarda lo standard input, si tratta invece di un flusso di byte di tipo `InputStream`. Il riferimento a questo stream è conservato in un campo statico denominato `in` della classe `System` del package `java.lang`.

A titolo d'esempio presentiamo un'applicazione che legge un intero da tastiera e lo stampa a video:

```
import java.io.*;

public class TastieraVideo {

 public static void main(String[] args) throws IOException {
 //stream per la lettura
 BufferedReader br = new BufferedReader(new
 InputStreamReader(System.in));

 System.out.print("Inserisci un intero: ");
 try {
 int i = Integer.parseInt(br.readLine());
 System.out.println("L'intero inserito è: " + i);
 }
 catch (NumberFormatException e) {
 System.out.println("La stringa inserita non rappresenta " +
 "un numero intero");
 }
 }
}
```

Si osservi che per leggere dati dalla tastiera, modellata da uno stream di byte, bisogna collegare l'oggetto `InputStream` cui fa riferimento `System.in` a uno stream di tipo `InputStreamReader`, che fornisce una traduzione da uno stream di byte a uno stream di caratteri; successivamente bisogna collegare quest'ultimo a un `BufferedReader`, che fornisce un metodo per la lettura di una stringa di caratteri, e quindi convertire la stringa ottenuta nel dato desiderato, che nell'esempio è un `int`.

La classe `PrintStream`, come altre classi che realizzano stream di output, mette a disposizione un metodo con un numero variabile di argomenti per scrivere stringhe specificando il formato da utilizzare. Nel caso di `PrintStream` il metodo è:

- `public PrintStream printf(String format, Object... args)`

Scrive sul `PrintStream` che esegue il metodo la stringa che rappresenta gli argomenti `args`, secondo il formato specificato dalla stringa `format` fornita come primo argomento (il metodo restituisce convenzionalmente un riferimento all'oggetto stesso che esegue il metodo, ma questa informazione non è in genere utilizzata).

La *stringa di formattazione* è una stringa che può contenere testo prefissato e uno o più *specificatori di formato*.

A titolo d'esempio, consideriamo le seguenti variabili:

```
int unità = 4;
String descrizione = "Latte";
double prezzo = 1.18;
```

che memorizzano le informazioni su un prodotto acquistato e immaginiamo di voler stampare tali informazioni aggiungendo il prezzo totale. Con l'istruzione:

```
System.out.printf("%2d - %-10s - %6.2f\n %2$24s: tot. %6.2f \n",
 unità, descrizione, prezzo, prezzo * unità);
```

L'effetto ottenuto è il seguente:

```
4 - Latte - 1.18
 Latte: tot. 4.72
```

Nell'istruzione precedente il metodo `printf` è stato invocato con 4 argomenti oltre alla stringa di formattazione. Quest'ultima indica, mediante degli *specificatori di formato*, come le informazioni specificate dagli argomenti devono essere “assemblate” per costruire la stringa da scrivere sullo stream. Gli specificatori di formato sono introdotti dal carattere %. La corrispondenza fra gli specificatori di formato e gli argomenti è, a meno di diverse indicazioni, *posizionale*; quindi il primo specificatore (nel nostro caso %2d) si applica al primo argomento (`unità`), il secondo (%-10s) al secondo argomento (`descrizione`) e così via. Ogni specificatore indica le regole con cui costruire una parte della stringa.

Il formato generale degli specificatori per gli argomenti di tipo numerico, carattere e `String` è il seguente:

$$\%[indice\_argomento\$][flags][ampiezza][.precisione]rappresentazione$$

I vari elementi che compongono uno specificatore di formato sono scritti fra parentesi quadre per indicare che sono opzionali.

Consideriamo il primo specificatore che compare nel nostro esempio, cioè %2d: esso indica solo l'ampiezza della stringa risultante, e il formato di rappresentazione dell'argomento.

L'*ampiezza* indica la lunghezza della stringa che dev'essere costruita. La *rappresentazione* è indicata da un carattere, e specifica come dev'essere rappresentato l'argomento. Ad esempio:

- d indica che l'argomento (che dev'essere un'espressione di tipo intero) dev'essere rappresentato come un intero decimale;
- o indica che l'argomento (che dev'essere un'espressione di tipo intero) dev'essere rappresentato in notazione ottale;
- s indica che l'argomento dev'essere rappresentato come una stringa;

- **f** indica che l'argomento (che dev'essere un'espressione di tipo `float` o `double`) dev'essere rappresentato in notazione decimale;
- **e** indica che l'argomento (che dev'essere un'espressione di tipo `float` o `double`) dev'essere rappresentato in notazione scientifica.

Quindi, tornando all'esempio, la specifica di formattazione `%2d` indica che il primo argomento deve essere rappresentato in notazione decimale, e che la stringa risultante deve essere di lunghezza due. Dato che il valore dell'espressione `unità` è 4, la stringa viene costruita facendo precedere al valore uno spazio bianco (cioè la rappresentazione dell'argomento viene allineata a destra).

I *flag* specificano modifiche del comportamento standard dello specificatore di formato. Ad esempio, il secondo specificatore che compare nel nostro esempio, `%-10s`, contiene il *flag* `-` che indica che la rappresentazione della stringa dev'essere *allineata a sinistra* nella stringa di lunghezza 10 descritta dallo specificatore. Si noti che nella stringa di formato dell'esempio ci sono due caratteri `-` che non si trovano all'interno di specificatori; essi vengono riportati nella stringa risultante così come gli spazi che li precedono e li seguono.

La *precisione*, che può essere indicata solo se l'argomento è un'espressione di tipo `float` o `double`, indica quante cifre della parte decimale del valore devono essere incluse nella stringa. Nel nostro esempio `%6.2f`, indica che il prezzo dev'essere inserito in una stringa di lunghezza 6 in cui sono rappresentate solo le prime due cifre della parte decimale.

Come abbiamo detto, di norma la corrispondenza fra gli specificatori di formato e gli argomenti è posizionale; è però possibile far riferimento a un argomento diverso indicando, all'inizio dello specificatore, l'indice dell'argomento da utilizzare seguito dal carattere `$`. Nel nostro esempio lo specificatore `%2$24s` costruisce una stringa di 24 caratteri utilizzando di nuovo il secondo argomento, cioè `descrizione`; si noti che l'indice del primo argomento è 1.

Osserviamo che se la stringa di formattazione non ha la corretta sintassi o non corrisponde agli argomenti forniti, il metodo solleva l'eccezione non controllata `IllegalFormatException` definita nel package `java.util`.

Qui ci siamo limitati a illustrare alcuni degli elementi utilizzati per la definizione delle stringhe di formattazione, per una presentazione dettagliata rimandiamo il lettore alla documentazione della classe `Formatter` del package `java.util`. Per concludere osserviamo che le stringhe di formattazione sono utilizzate in varie classi delle API; ad esempio, la classe `String` fornisce un metodo statico `String format(String format, Object... args)` che restituisce la stringa costruita combinando gli argomenti secondo le regole descritte nella stringa `formato`.

## 13.8 Lettura e scrittura di oggetti su stream di byte

In molte applicazioni risulta utile scambiare oggetti in formato binario. A questo scopo Java fornisce strumenti per scrivere oggetti su uno stream di byte e leggere oggetti da uno stream di byte. Per farlo è necessario trasformare un oggetto in una sequenza di byte e ricostruire un oggetto a partire da una sequenza di byte. Il processo di conversione di un oggetto in un flusso di byte prende il nome di *serializzazione*, mentre il processo inverso, di ricostruzione di un oggetto

a partire da un flusso di byte, è detto *deserializzazione*. Le classi che forniscono le funzionalità per serializzare e deserializzare oggetti sono `ObjectOutputStream` e `ObjectInputStream`.

La classe `ObjectOutputStream` fornisce metodi per convertire un oggetto in una sequenza di byte allo scopo di inviarlo a un `OutputStream`. La classe dispone del seguente costruttore:

- `public ObjectOutputStream(OutputStream out) throws IOException`  
Crea un `ObjectOutputStream` che scrive sull'`OutputStream` specificato.

Il metodo principale della classe è:

- `public final void writeObject(Object obj) throws IOException`  
Scrive l'oggetto specificato come argomento sull'`ObjectOutputStream` che esegue il metodo. L'oggetto fornito come argomento deve implementare l'interfaccia `Serializable`; in caso contrario verrà sollevata l'eccezione non controllata `NotSerializableException`.

La classe mette a disposizione metodi `write` per ogni tipo primitivo con la stessa segnatura di quelli forniti dalla classe `DataOutputStream`.

Le funzionalità necessarie per deserializzare un oggetto sono fornite dalla classe `ObjectInputStream`. Tale classe mette a disposizione il costruttore:

- `public ObjectInputStream(InputStream in) throws IOException`  
Crea un `ObjectInputStream` che legge dall'`InputStream` specificato.

Il metodo principale della classe è:

- `public final Object readObject() throws IOException, ClassNotFoundException`  
Legge un oggetto dall'`ObjectInputStream` che esegue il metodo.

Inoltre fornisce metodi `read` per ogni tipo primitivo con la medesima segnatura di quelli forniti dalla classe `DataInputStream`.

Prima di analizzare in dettaglio i metodi di lettura e scrittura di oggetti presentiamo un esempio. Consideriamo la seguente applicazione che scrive su un file una coppia di oggetti costituita da una stringa e da una data, quindi la rilegge dal file e la visualizza.

```
import java.io.*;
import prog.utili.Data;

public class ScriviLeggi {

 public static void scrivi() throws IOException {
 FileOutputStream fos = new FileOutputStream("dataOdierna");
 ObjectOutputStream oos = new ObjectOutputStream(fos);
 oos.writeObject("Data di oggi: ");
 oos.writeObject(new Data());
 }
}
```

```

 oos.flush();
 oos.close();
 fos.close();
}

public static void leggi() throws IOException,
 ClassNotFoundException {
 FileInputStream fis = new FileInputStream("dataOdierna");
 ObjectInputStream ois = new ObjectInputStream(fis);
 String s = (String)ois.readObject();
 Data d = (Data)ois.readObject();
 ois.close();
 fis.close();

 System.out.println(s + d.toString());
}

public static void main(String[] args) throws IOException,
 ClassNotFoundException {
 //scrittura
 scrivi();
 //lettura dell'oggetto
 leggi();
}
}

```

Nel metodo `scrivi` viene creato un `ObjectOutputStream` collegato a un file denominato "dataOdierna", su cui vengono scritti due oggetti: una stringa e un oggetto della classe `Data`. Come si desume dalla documentazione, ambedue le classi, `String` e `Data`, implementano l'interfaccia `Serializable` secondo il contratto del metodo `writeObject`, e quindi gli oggetti vengono scritti su file. Ovviamente il contenuto del file, trattandosi di un file binario e non di un file di caratteri, non è direttamente intelligibile e include la rappresentazione binaria dei due oggetti memorizzati.

Il metodo `leggi` effettua la lettura dei due oggetti e li visualizza. Si osservi che la lettura dei due oggetti viene effettuata mediante due `readObject`, senza specificare il tipo dell'oggetto da leggere. Il meccanismo di deserializzazione è in grado di ricostruire l'oggetto correttamente a patto di disporre della classe di cui l'oggetto è istanza, cioè a patto che la Java Virtual Machine possa accedere al bytecode della classe di cui l'oggetto è istanza.

Abbiamo detto che per poter serializzare gli oggetti di una classe, questa deve implementare l'interfaccia `Serializable` definita nel package `java.io`. Si tratta di un'interfaccia priva di metodi, definita nella terminologia di Java *interfaccia marker*. Dichiarendo che una classe implementa l'interfaccia `Serializable`, sostanzialmente dichiariamo di consentire la serializzazione

dei suoi oggetti. Proviamo a vedere che cosa accade se tentiamo di serializzare un oggetto di una classe che non implementa l'interfaccia `Serializable`. Consideriamo questa classe:

```
public class Alieno {
 private String alieno;

 public Alieno(String s){
 alieno = s;
 }
}
```

e consideriamo la seguente applicazione che scrive un'istanza di `Alieno` su file:

```
import java.io.*;

public class ScriviAlieno {
 public static void main(String[] args) throws IOException,
 ClassNotFoundException {
 //scrittura dell'oggetto
 FileOutputStream fos = new FileOutputStream("fileAlieno");
 ObjectOutputStream oos = new ObjectOutputStream(fos);
 oos.writeObject(new Alieno("Et"));
 oos.flush();
 oos.close();
 fos.close();
 }
}
```

Quest'applicazione viene compilata correttamente, ma eseguendola si verifica l'eccezione `NotSerializableException` definita nel package `java.io`:

```
> java ScriviAlieno
Exception in thread "main" java.io.NotSerializableException: Alieno
 at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1054)
 at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:278)
 at ScriviAlieno.main(ScriviAlieno.java:9)
```

L'eccezione viene sollevata nel metodo `writeObject` della classe `ObjectOutputStream`. All'interno del metodo `writeObject` si verifica se l'oggetto `o`, passato come argomento, implementa l'interfaccia `Serializable` con un'istruzione del tipo:

```
if (!(o instanceof Serializable))
 throw NotSerializableException();
```

Per ragioni di sicurezza le classi non sono automaticamente serializzabili. Una volta che un oggetto è stato serializzato ed eventualmente scritto su file, non esistono più, infatti, i meccanismi forniti in fase di esecuzione per garantire la protezione delle informazioni memorizzate nell'oggetto. Modificando la classe `Alieno` in modo che implementi l'interfaccia `Serializable` come segue:

```
import java.io.Serializable;

public class Alieno implements Serializable {
 private String alieno;

 public Alieno(String s) {
 alieno = s;
 }

 public String toString() {
 return alieno;
 }
}
```

L'applicazione precedente viene eseguita correttamente.

Per quel che concerne la deserializzazione di un oggetto, abbiamo detto che la classe di cui l'oggetto è istanza dev'essere disponibile. Per la precisione, al fine di ricostruire correttamente la struttura dell'oggetto letto, la Java Virtual Machine deve poter accedere a tutte le classi che sono state coinvolte nel processo di serializzazione. Se ad esempio proviamo a eseguire l'applicazione `LeggiAlieno` in una directory che contiene il file `fileAlieno`, ma non il bytecode della classe `Alieno`, si verifica questa eccezione:

```
> java LeggiAlieno
Exception in thread "main" java.lang.ClassNotFoundException: Alieno
 at java.net.URLClassLoader$1.run(URLClassLoader.java:200)
 at java.security.AccessController.doPrivileged(Native Method)
 at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
 at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
 at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:276)
 at java.lang.ClassLoader.loadClass(ClassLoader.java:251)
 at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:319)
 at java.lang.Class.forName0(Native Method)
 at java.lang.Class.forName(Class.java:247)
 at java.io.ObjectInputStream.resolveClass(ObjectInputStream.java:604)
 at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:
 1575)
 at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1496)
```

```
at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1732)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1329)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:351)
at LeggiAlieno.main(LeggiAlieno.java:9)
```

Per concludere, osserviamo che il processo di serializzazione di un singolo oggetto potrebbe coinvolgere più oggetti. Ad esempio, nel caso della serializzazione di un oggetto della classe `Alieno`, salvare un'istanza di `Alieno` significa salvare anche un'istanza dell'oggetto di tipo `String` il cui riferimento è memorizzato nel campo `alieno` dell'oggetto. In generale la rete dei riferimenti che si dirama da un oggetto può essere arbitrariamente complicata e contenere anche cicli. Nel processo di serializzazione Java applica un algoritmo che trasforma la rete di oggetti in una sequenza di oggetti che vengono quindi salvati; il nome serializzazione si riferisce proprio a quest'operazione. Come esemplificato dalle seguenti applicazioni, è possibile scrivere e leggere direttamente un oggetto complesso come una `LinkedList` a patto che gli oggetti memorizzati nella lista siano tutti istanze di classi che implementano l'interfaccia `Serializable`. L'applicazione `ScriviListaStringhe` scrive in un file una `LinkedList` che memorizza una sequenza di stringhe letta da tastiera.

```
import java.io.*;
import java.util.LinkedList;

public class ScriviListaStringhe {

 public static void main(String[] args) throws IOException {
 //predisposizione del canale di lettura dalla tastiera
 BufferedReader br = new
 BufferedReader(new InputStreamReader(System.in));

 LinkedList<String> lista = new LinkedList<String>();
 String s;
 System.out.print("Inserisci una stringa (^D per terminare): ");
 while ((s = br.readLine()) != null) {
 lista.add(s);
 System.out.print("Inserisci una stringa (^D per terminare): ");
 }
 //predisposizione del canale di scrittura sul file
 FileOutputStream fos = new FileOutputStream("listaStringhe");
 ObjectOutputStream oos = new ObjectOutputStream(fos);
 oos.writeObject(lista);
 }
}
```

L'applicazione LeggiListaStringhe legge invece la `LinkedList` memorizzata nel file e ne visualizza gli elementi a video.

```
import java.io.*;
import java.util.LinkedList;
import java.util.Iterator;

public class LeggiListaStringhe {

 public static void main(String[] args) throws IOException,
 ClassNotFoundException {
 //predisposizione del canale di lettura dal file
 FileInputStream fis = new FileInputStream("listaStringhe");
 ObjectInputStream ois = new ObjectInputStream(fis);

 LinkedList<String> lista = (LinkedList<String>)ois.readObject();
 for (String s: lista)
 System.out.println(s);
 }
}
```

# **Parte IV**

# **Appendici**

## Appendice A

# Tipi primitivi

Il linguaggio Java dispone di tipi primitivi per la rappresentazione di numeri interi, numeri in virgola mobile, valori booleani e caratteri. A differenza di quanto accade in altri linguaggi di programmazione, dove la quantità di memoria occupata da una variabile di tipo primitivo e l'insieme dei valori (*range*) rappresentabili da questa vengono stabiliti da chi implementa il compilatore sulla base delle caratteristiche dell'architettura sottostante, in Java l'insieme dei valori dei tipi primitivi è fissato con il linguaggio.

I tipi primitivi di Java sono i seguenti:

- `boolean`, per rappresentare un valore di verità;
- `char`, per rappresentare i caratteri;
- `byte`, `short`, `int` e `long`, per rappresentare numeri interi;
- `float` e `double`, per rappresentare numeri in virgola mobile.

La tabella sottostante riassume le caratteristiche dei tipi primitivi di Java, evidenziando il tipo di dato che rappresentano, la quantità di memoria occupata, il range dei valori, e infine il valore di default utilizzato per inizializzare i campi, i campi statici e le componenti degli array di tipo primitivo.

| Tipo                 | Valori                                   | Bit    | Range                                     | Default            |
|----------------------|------------------------------------------|--------|-------------------------------------------|--------------------|
| <code>boolean</code> | <code>true</code> , <code>false</code>   | -      | <code>true</code> e <code>false</code>    | <code>false</code> |
| <code>char</code>    | Caratteri Unicode                        | 16 bit | da \u0000 a \uFFFF                        | \u0000             |
| <code>byte</code>    | Numeri interi con segno                  | 8 bit  | da -2 <sup>7</sup> a 2 <sup>7</sup> - 1   | 0                  |
| <code>short</code>   | Numeri interi con segno                  | 16 bit | da -2 <sup>15</sup> a 2 <sup>15</sup> - 1 | 0                  |
| <code>int</code>     | Numeri interi con segno                  | 32 bit | da -2 <sup>31</sup> a 2 <sup>31</sup> - 1 | 0                  |
| <code>long</code>    | Numeri interi con segno                  | 64 bit | da -2 <sup>63</sup> a 2 <sup>63</sup> - 1 | 0                  |
| <code>float</code>   | Numeri floating point, standard IEEE 754 | 32 bit | da ± 1.4E-45 a ± 3.4E+38                  | 0.0                |
| <code>double</code>  | Numeri floating point, standard IEEE 754 | 64 bit | da ± 4.9E-324 a ± 1.8E+308                | 0.0                |

Le sequenze di caratteri utilizzate per indicare direttamente nel codice valori di un tipo vengono chiamate in Java *letterali*. Ad esempio la sequenza di caratteri 100 è un letterale di tipo `int` che rappresenta il numero intero 100; 'a' è invece un letterale per il tipo `char` che rappresenta la lettera a. In questa appendice presenteremo alcuni dettagli sui tipi interi, il tipo `char` e la tabella completa degli operatori del linguaggio.

## A.1 Tipi interi

I *tipi interi* in Java rappresentano *numeri interi con segno*, e sono quattro: `byte`, `short`, `int` e `long`.

I letterali di tipo intero vengono specificati utilizzando la solita rappresentazione decimale mediante sequenze di cifre decimali. Ad esempio 35 e 12233445 sono letterali di tipo intero. È possibile specificare i letterali di tipo intero anche tramite la notazione *ottale* e la notazione *esadecimale*.

I letterali di tipo `long` vengono specificati utilizzando la notazione per i letterali interi seguita dal carattere L o dal carattere l (è preferibile il carattere maiuscolo perché ben distinto dalla cifra 1). Se un letterale di tipo intero non è seguito dal carattere L o dal carattere l, allora è considerato di tipo `int`. Quindi 0 e 100 sono letterali di tipo `int`, mentre 0L e 100L sono letterali di tipo `long`. Gli assegnamenti:

```
int i;
i = 0L;
i = 100L;
```

non sono dunque legali. Invece gli assegnamenti:

```
long l;
l = 0;
l = 100;
```

lo sono; in questo caso il letterale viene promosso a `long` prima dell'assegnamento.

Se un letterale intero rappresenta un numero intero più grande di 2147483647 ( $2^{31} - 1$ ) o più piccolo di -2147483648 ( $-2^{31}$ ) e viene assegnato a una variabile di tipo `int`, si verifica un errore in fase di compilazione. Tutti i letterali che rappresentano numeri compresi fra 0 e 2147483647 ( $2^{31} - 1$ ) possono essere utilizzati ovunque si possa usare un `int`, mentre il letterale 2147483648 ( $2^{31}$ ) può comparire solo come operando dell'operatore unario -.

Se un letterale intero rappresenta un numero intero maggiore di 9223372036854775807 ( $2^{63}-1$ ) o minore di -9223372036854775808 ( $-2^{63}$ ) e viene assegnato a una variabile di tipo `long`, si verifica un errore in fase di compilazione. Tutti i letterali che rappresentano numeri compresi fra 0 e 2147483647 ( $2^{63}-1$ ) possono essere utilizzati ovunque si possa utilizzare un `int`, mentre il letterale 2147483648 ( $2^{63}$ ) può comparire solo come operando dell'operatore unario -.

In Java l'aritmetica intera è *modulare*, quindi non viene mai prodotto un *overflow* quando si eccede il range rappresentabile per un dato tipo intero. Ad esempio, nel caso delle istruzioni:

```
int i = 2147483647; // = 2^31
i = i + 1; // = 2^31+1
System.out.println(i);
```

il valore stampato è -2147483648.

Infine osserviamo che la divisione intera per zero produce un errore in fase di esecuzione, di tipo `ArithmetricException`.

## A.2 Il tipo char

In Java le variabili di tipo `char` hanno una dimensione di 16 bit allo scopo di rappresentare uno dei possibili caratteri del codice *Unicode*, una codifica di caratteri dell'alfabeto internazionale a 16 bit. I letterali di tipo `char` sono costruiti racchiudendo fra apici singoli (') un carattere ASCII o *sequenze di escape*. Le sequenze di escape sono sequenze di caratteri ASCII che iniziano con il carattere speciale \. Ad esempio i caratteri dell'alfabeto Unicode possono essere rappresentati da sequenze di escape della forma `\uHHHH`, dove ogni carattere H nella sequenza indica una cifra esadecimale, cioè uno dei caratteri da 0 a 9 o da A a F (maiuscole o minuscole). Ad esempio, la lettera A è rappresentata dalla sequenza `\u0041`, e quindi la definizione:

```
char c = '\u0041';
```

dichiara `c` come una variabile di tipo `char` e le assegna il valore `\u0041`, che rappresenta appunto la lettera A. Lo stesso effetto si ottiene con la definizione:

```
char c = 'A';
```

in cui viene utilizzato il carattere ASCII.

Nella seguente tabella sono riportate altre sequenze di escape utili nella scrittura dei programmi. Alcune rappresentano caratteri stampabili, altre caratteri speciali.

| Sequenza di escape | Unicode | Significato        |
|--------------------|---------|--------------------|
| \b                 | \u0008  | backspace BS       |
| \t                 | \u0009  | horizontal tab HT  |
| \n                 | \u000a  | linefeed LF        |
| \f                 | \u000c  | form feed FF       |
| \r                 | \u000d  | carriage return CR |
| \"                 | \u0022  | double quote ("")  |
| \'                 | \u0027  | single quote (')   |
| \\\                | \u005c  | backslash          |

Effettuando ad esempio l'assegnamento: `c = '\n'` e stampando a video la variabile `c` si ottiene l'effetto di mandare a capo il cursore. I caratteri ' e \ sono invece rappresentati da sequenze di escape perché hanno un significato speciale e non possono essere utilizzate direttamente all'interno di un programma Java (ad esempio ' indica l'inizio e la fine di un letterale carattere). Quindi, se vogliamo ad esempio assegnare a `c` il carattere ', dovremo utilizzare la sequenza di escape \' scrivendo `c = '\''`.

## A.3 Operatori

In conclusione riportiamo la seguente tabella riassuntiva degli operatori del linguaggio Java. Gli operatori sono indicati in ordine di precedenza, dalla più alta alla più bassa; gli operatori che compaiono nello stesso riquadro hanno la medesima precedenza. Nella colonna centrale sono indicati, separati da virgolette, gli argomenti previsti dall'operatore, mentre nella colonna a destra si evidenzia il significato degli operatori stessi.

| Operatore     | Tipo degli operandi                    | Operazione                                         |
|---------------|----------------------------------------|----------------------------------------------------|
| .             | oggetto, membro                        | accesso a membro dell'oggetto                      |
| [ ]           | array, int                             | accesso a elemento di array                        |
| (args)        | argomenti                              | invocazione di metodo                              |
| ++, --        | variabile                              | incremento e decremento postfissi                  |
| ++, --        | variabile                              | incremento e decremento prefissi                   |
| +, -          | numeri                                 | più e meno unari                                   |
| !             | boolean                                | NOT                                                |
| new<br>(type) | classe, elenco args<br>tipo, qualsiasi | creazione di oggetti<br>cast o conversione di tipo |
| *, /, %       | numero, numero                         | moltiplicazione, divisione e resto                 |
| +, -          | numero, numero                         | addizione, sottrazione                             |
| +             | stringa, qualsiasi                     | concatenazione di stringhe                         |
| <<, >>        | intero, intero                         | shift di bit                                       |
| <, <=, >, >=  | numero, numero                         | confronto                                          |
| instanceof    | riferimento, classe o interfaccia      | comparazione del tipo                              |
| ==, !=        | primitivo, primitivo                   | confronto (sui valori)                             |
| ==, !=        | riferimento, riferimento               | confronto (sui riferimenti)                        |
| &             | boolean, boolean<br>intero, intero     | AND                                                |
| ^             | boolean, boolean<br>intero, intero     | OR esclusivo                                       |
|               | boolean, boolean<br>intero, intero     | OR                                                 |
| &&            | boolean, boolean                       | AND (lazy)                                         |
|               | boolean, boolean                       | OR (lazy)                                          |
| ? :           | boolean, qualsiasi, qualsiasi          | condizione                                         |
| =             | variabile, qualsiasi                   | assegnamento                                       |

Tutti gli operatori binari, tranne l'operatore di assegnamento, sono associativi a sinistra. L'operatore di assegnamento e l'operatore condizionale ?: sono associativi a destra.

# Strutture di controllo

In questa appendice riepiloghiamo la sintassi e la semantica delle strutture di controllo del linguaggio Java.

## B.1 L'istruzione if-else

Nella sua forma più generale l'istruzione `if-else` ha questa sintassi:

```
if (condizione)
 istruzione1
else
 istruzione2
```

Dove:

- *condizione* è una qualunque espressione che restituisce un valore di tipo `boolean` scritta obbligatoriamente tra parentesi tonde;
- *istruzione1* e *istruzione2* sono istruzioni singole oppure *blocchi di istruzioni*, cioè sequenze di istruzioni racchiuse tra parentesi graffe.

L'esecuzione di un'istruzione `if-else` avviene come segue.

- (1) Per prima cosa viene valutata la condizione; tale valutazione restituisce uno dei due valori, `true` o `false`.
- (2) A questo punto l'esecuzione prosegue in modo diverso a seconda che il valore della condizione sia `true` o `false`:
  - se la condizione risulta vera (cioè se il suo valore è `true`), viene eseguita *istruzione1*;
  - se invece la condizione è falsa (cioè se il suo valore è `false`), viene eseguita *istruzione2*.
- (3) L'esecuzione riprende dalla prima istruzione dopo l'istruzione `if-else`.

Il ramo `else` dell’istruzione può essere omesso: in questo caso si parla semplicemente di istruzione `if`. Il suo schema è:

```
if (condizione)
 istruzioneI
```

L’esecuzione procede come segue.

- (1) Per prima cosa viene valutata l’espressione che costituisce la condizione.
- (2) A questo punto l’esecuzione prosegue in modo diverso a seconda che il valore della condizione sia `true` o `false`:
  - se la condizione risulta vera (cioè se il suo valore è `true`), viene eseguita *istruzioneI*
  - se invece la condizione è falsa (cioè se il suo valore è `false`), l’esecuzione riprende dalla prima istruzione dopo l’istruzione `if`.

## B.2 L’istruzione `while`

L’istruzione `while` ha questa forma:

```
while (condizione)
 istruzione
```

Dove:

- *condizione* è un’espressione booleana scritta obbligatoriamente tra parentesi tonde
- *istruzione* è l’istruzione che dev’essere ripetuta: può essere un’istruzione singola oppure un blocco di istruzioni.

L’esecuzione di un’istruzione `while` procede come segue.

- (1) Viene valutata l’espressione *condizione*; trattandosi di un’espressione booleana, tale valutazione restituisce uno dei due valori, `true` o `false`.
- (2) Se la condizione è vera (il suo valore è `true`):
  - viene eseguita l’istruzione nel corpo del ciclo;
  - l’esecuzione continua dal Punto (1).
- (3) Se la condizione è falsa (il suo valore è `false`), il ciclo termina e l’esecuzione continua dalla prima istruzione dopo il ciclo `while`.

Quindi, l’esecuzione del ciclo termina quando *condizione* risulta falsa. Inoltre, poiché il controllo avviene in testa, *istruzione* può essere eseguita anche zero volte.

## B.3 L'istruzione do...while

Il ciclo `do...while` è una variante del ciclo `while` in cui la condizione viene valutata dopo l'esecuzione del corpo del ciclo anziché prima. L'istruzione `do-while` ha questa forma:

```
do
 istruzione
 while (condizione)
```

Dove:

- *condizione* è un'espressione di tipo boolean;
- *istruzione* è l'istruzione che dev'essere ripetuta: può essere un'istruzione singola oppure un blocco di istruzioni.

L'esecuzione avviene come segue.

- (1) Viene eseguito il corpo del ciclo, cioè *istruzione*.
- (2) Viene valutata l'espressione *condizione*.
- (3) Se la condizione è vera (il suo valore è `true`), l'esecuzione prosegue dal Punto (1).
- (4) Se la condizione è falsa (il suo valore è `false`), il ciclo termina e l'esecuzione riprende dalla prima istruzione dopo l'istruzione `do-while`.

Si osservi che, come nel caso del ciclo `while`, l'esecuzione del ciclo termina quando la condizione risulta falsa. Diversamente dal ciclo `while`, il cui corpo può non essere eseguito, il corpo del ciclo `do...while` viene sempre eseguito almeno una volta.

## B.4 L'istruzione for

La forma dell'istruzione `for` è:

```
for (expr_inizializzazione ; condizione ; expr_incremento)
 istruzione
```

Dove:

- *expr\_inizializzazione* è una lista di espressioni, separate da virgola; sono le espressioni di inizializzazione delle variabili di controllo del ciclo e possono eventualmente contenere anche la dichiarazione delle variabili;
- *condizione* è una qualunque espressione booleana;
- *expr\_incremento* è una lista di espressioni;

- *istruzione* è una singola istruzione oppure un blocco di istruzioni.

Tutte le componenti di un ciclo `for`, cioè *espr\_inizializzazione*, *condizione*, *espr\_incremento* e *istruzione* sono opzionali. L'esecuzione del ciclo avviene come segue.

- (1) Per prima cosa vengono valutate le espressioni che compaiono in *espr\_inizializzazione*.
- (2) Quindi viene valutata l'espressione *condizione*; trattandosi di un'espressione booleana, tale valutazione restituisce uno dei due valori, `true` o `false`.
- (3) Se la condizione è vera (il suo valore è `true`):
  - viene eseguito il blocco di istruzioni nel corpo del ciclo;
  - vengono valutate le espressioni che compaiono in *espr\_incremento*;
  - l'esecuzione prosegue dal Punto (2).
- (4) Se la condizione è falsa (il suo valore è `false`), l'esecuzione riprende dalla prima istruzione dopo l'istruzione `for`.

## L'istruzione for-each

L'istruzione `for-each` è un'istruzione di iterazione appositamente disegnata per scandire array o oggetti che implementino l'interfaccia generica `Iterable<E>`. La forma dell'istruzione è:

```
for (Tipo identificatore : espressione)
 istruzione
```

Nel caso venga utilizzata per scandire un array:

- *espressione* dev'essere un'espressione che restituisce il riferimento a un array;
- *Tipo identificatore* è la dichiarazione della variabile nella quale vengono memorizzati di volta in volta gli elementi dell'array e *Tipo* dev'essere un tipo a cui possono essere assegnati gli elementi dell'array.

In questo caso il ciclo `for-each` risulta equivalente a:

```
for (int i = 0; i < espressione.length; i++) {
 Tipo identificatore = espressione[i];
 istruzione
}
```

Nel caso in cui venga utilizzata per scandire oggetti che implementano l'interfaccia `Iterable<E>`:

- *espressione* dev'essere un'espressione che restituisce il riferimento a un oggetto che implementa l'interfaccia `Iterable<E>`;

- *Tipo identificatore* è la dichiarazione della variabile nella quale vengono memorizzati di volta in volta gli elementi prelevati dall'iteratore e *Tipo* dev'essere un supertipo di *E* o *E* stesso.

In questo caso il ciclo for-each risulta equivalente a:

```
for (Iterator<E> i = espressione.iterator(); i.hasNext();) {
 Tipo identificatore = i.next();
 istruzione
}
```

## B.5 L'istruzione switch

L'istruzione *switch* consente di eseguire una selezione tra diverse istruzioni in base al risultato di un'espressione intera. Lo schema di tale istruzione è:

```
switch (espressione)
 bloccoSwitch
```

dove *bloccoSwitch* ha la seguente struttura:

```
{
 case val1:
 blocco1
 case val2:
 blocco2
 case valN:
 bloccoN
 default:
 bloccoDefault
}
```

e:

- *espressione* dev'essere un'espressione di tipo *char*, *byte*, *short* o *int* oppure di un tipo enumerativo, obbligatoriamente scritta tra parentesi; tale espressione è chiamata *selettore*
- *val1*, ..., *valN* prendono il nome di *etichette* e devono essere costanti assegnabili al tipo del selettore;
- *blocco1*, ..., *bloccoN* e *bloccoDefault* sono sequenze (anche vuote) di istruzioni;
- il caso *default* è opzionale.

L'esecuzione dell'istruzione *switch* avviene selezionando un punto di inizio nel *bloccoSwitch* secondo questo procedimento.

- (1) Viene valutata *espressione*.
- (2) Se c'è un'etichetta con associata una costante con valore uguale al risultato dell'espressione, si inizia a eseguire il corpo dell'istruzione **switch** a partire da tale etichetta. Se essa non c'è, ma è presente un'etichetta **default**, si inizia a eseguire il blocco di codice a partire da questa.
- (3) Se non c'è un'etichetta con associata una costante con valore uguale a quello calcolato e nemmeno un'etichetta **default**, si passa a eseguire il codice che si trova dopo l'istruzione **switch**.
- (4) Se durante l'esecuzione del *bloccoSwitch*, a partire dal punto selezionato in base alle etichette, si incontra un'istruzione **break**, si passa a eseguire il codice che si trova dopo l'istruzione **switch**.

Come evidenziato nel Punto (4), le etichette **case** e **default** non provocano la terminazione dell'esecuzione dello **switch**. Di conseguenza, se viene eseguito il blocco associato a un **case**, una volta terminato il blocco, l'esecuzione prosegue con le istruzioni del blocco **case** successivo. Questo salvo che non sia presente l'istruzione **break**, il cui effetto è interrompere l'esecuzione del blocco di istruzioni corrente (in questo caso il corpo dell'istruzione **switch**) e di far riprendere l'esecuzione dalla prima istruzione dopo l'istruzione **switch**.

## B.6 Le istruzioni **break** e **continue**

L'istruzione **break** può essere utilizzata per concludere l'esecuzione di un blocco di codice. Può essere usata solo in un'istruzione **switch**, **for**, **while** e **do...while**, e il suo effetto è quello di terminarne l'esecuzione. Abbiamo già visto l'effetto dell'istruzione **break** nell'istruzione **switch**; per quel che riguarda i cicli, l'istruzione **break** forza la terminazione del ciclo più interno in esecuzione.

Si può utilizzare l'istruzione **break** con un'etichetta nel modo seguente:

**break** *etichetta*;

dove *etichetta* è un identificatore e può essere associata a una qualunque istruzione scrivendo:

*etichetta*: *istruzione*

Le etichette possono essere impiegate unicamente dalle istruzioni **break** e **continue**. L'istruzione:

**break** *e*

all'interno dell'istruzione etichettata da *e* provoca la terminazione di tale istruzione. Questa forma di **break** può essere usata per forzare la terminazione di più cicli innestati. Ad esempio il seguente schema:

```

e: for (... ; ...; ...) {
 ...
 for (...; ...;) {
 ...
 if (condizione)
 break e;
 ...
 }
 ...
}

```

consente di forzare la terminazione dei due cicli **for** innestati se **condizione** risulta vera.

L'istruzione **continue** può essere utilizzata solo all'interno di un ciclo, cioè solo in un'istruzione **for**, **while** o **do...while**, e il suo effetto è quello di concludere l'esecuzione dell'iterazione corrente. Nel caso dei cicli **while** e **do-while**, l'esecuzione riprende dalla valutazione della condizione del ciclo. Nel caso del ciclo **for**, l'esecuzione riprende dalla valutazione delle espressioni di incremento. Usata con un'etichetta, nella forma:

```
continue etichetta;
```

termina l'esecuzione dell'iterazione del ciclo cui è associata l'etichetta.

Si osservi che le istruzioni **break** e **continue** con etichetta realizzano istruzioni di salto in cui sono dati forti vincoli sulla posizione dell'etichetta. Queste limitazioni hanno lo scopo di evitare il più possibile la confusione nel flusso di controllo di un programma dovuta a un uso indiscriminato dei salti. Nonostante questo, tali istruzioni dovrebbero essere utilizzate con estrema parsimonia, ossia solo quando il loro impiego semplifica notevolmente la struttura del codice.

## B.7 L'istruzione **return**

L'istruzione **return** permette di concludere l'esecuzione di un metodo ed eventualmente di restituire un valore al chiamante. Nel primo caso l'istruzione viene utilizzata nella forma:

```
return;
```

all'interno di un metodo che dichiari **void** come tipo del valore restituito. L'effetto dell'istruzione è quello di concludere l'esecuzione del metodo.

Qualora si abbia un metodo con un tipo restituito differente da **void**, l'istruzione dev'essere utilizzata nella forma:

```
return espressione;
```

dove *espressione* dev'essere di un tipo assegnabile al tipo restituito dal metodo. In un metodo che restituisca un tipo diverso da **void**, deve comparire un'istruzione **return** per ogni possibile flusso di esecuzione del metodo.

## B.8 L'istruzione try-catch

L'istruzione **try-catch** consente di intercettare eventuali eccezioni sollevate durante l'esecuzione delle istruzioni del blocco **try** e specifica un opportuno gestore per ognuna di esse. La forma dell'istruzione è la seguente:

```
try {
 // Codice che potrebbe generare eccezioni
 blocco_try
} catch (Tipo_Eccezione1 id1) { //Gestisce le eccezioni Tipo_Eccezione1
 blocco_gestore1
} catch (Tipo_Eccezione2 id2) { //Gestisce le eccezioni Tipo_Eccezione2
 blocco_gestore2
}
... finally {
 blocco_finally
}
```

Il codice in *blocco\_try* è il blocco di codice che si vorrebbe eseguire e che può sollevare eccezioni; ciascuna clausola **catch** costituisce invece il gestore di un determinato tipo di eccezione specificato come argomento della clausola. Nello schema descritto sopra il primo blocco **catch** è ad esempio preposto a gestire le eccezioni di tipo *Tipo\_Eccezione1*; l'identificatore *id1* specificato dopo il tipo dell'eccezione può essere utilizzato in riferimento all'oggetto eccezione intercettato dal gestore all'interno del codice *blocco\_gestore1*. Non esistono limiti al numero di gestori che possono essere presenti dopo un blocco **try**. Il blocco **finally** dell'istruzione è opzionale.

L'esecuzione avviene come segue.

- (1) Vengono eseguite le istruzioni nel blocco **try**; se durante l'esecuzione di questo codice non viene sollevata alcuna eccezione, l'esecuzione prosegue dal Punto (4).
- (2) Se durante l'esecuzione del codice nel blocco **try** viene sollevata un'eccezione, la Java Virtual Machine scorre la lista degli argomenti dei blocchi **catch**, nell'ordine in cui sono scritti, alla ricerca di un tipo cui possa essere ricondotto quello dell'eccezione sollevata (cioè che sia uguale o supertipo di quello dell'eccezione sollevata). Quando lo trova, esegue il codice del corrispondente blocco **catch**, al termine del quale passa al Punto (4).
- (3) Se nessuno dei blocchi **catch** è preposto a intercettare l'eccezione sollevata, vengono eseguite le istruzioni contenute nel blocco **finally** (quando tale blocco è presente), quindi l'esecuzione prosegue come se il blocco **try-catch** non fosse presente, rinviando cioè l'eccezione all'esterno del blocco **try-catch**.
- (4) Se il blocco **finally** non è presente, l'esecuzione riprende dalla prima istruzione dopo l'istruzione **try-catch**; in caso contrario vengono eseguite le istruzioni del blocco **finally**, e poi l'esecuzione continua con la prima istruzione che segue l'istruzione **try-catch**.

Si osservi che le istruzioni presenti nel blocco `finally` sono comunque eseguite, sia che venga completato il blocco `try` senza sollevare l'eccezione, sia che venga eseguito uno dei blocchi `catch`, sia che venga sollevata un'eccezione nel blocco `try` ma non venga eseguito alcun blocco `catch`. Inoltre le istruzioni indicate nel blocco `finally` saranno eseguite anche nel caso che all'interno di `try` o di `catch` il normale flusso di esecuzione venga interrotto da istruzioni come `return`, `break` e `continue`.