A1

Introduzione alla programmazione e alla progettazione orientate agli oggetti

Informatizzare una realtà significa crearne un modello astratto finalizzato alla gestione dei flussi informativi che la caratterizzano. L'ambiente circostante è costituito da sistemi spesso molto articolati: i mezzi di comunicazione, quelli di trasporto e le stesse realtà urbane in cui viviamo mostrano scenari in continua trasformazione, dove vari tipi di soggetti (persone, veicoli, mezzi di trasmissione, computer, ...) interagiscono tra di loro a diversi livelli di complessità. Studiare il comportamento di questi sistemi e delle loro interazioni per crearne un modello informatico richiede l'utilizzo di tecniche che siano al tempo stesso efficaci ed efficienti. Secondo una visione ormai condivisa e consolidata, un primo passo in questa direzione è rappresentato dall'individuazione e astrazione delle entità di riferimento per tali contesti.

Consideriamo a questo proposito un oggetto di uso quotidiano come un televisore, allo scopo di volerne creare un modello per descriverne sia gli aspetti statici (non soggetti a cambiamento) sia quelli dinamici (soggetti a cambiamento).

Alcuni dati (attributi) significativi sono:

- la dimensione in pollici dello schermo;
- il tipo di schermo (LCD, LED, OLED, ...);
- il colore esterno;
- il canale attualmente selezionato;
- il livello del volume impostato;
- il livello della luminosità regolato;
- ...

Il contenuto informativo che essi descrivono è relativo sia alle **caratteristiche** del televisore (colore esterno, dimensione in pollici, ...), sia allo **stato** in cui esso si trova in un determinato momento (acceso/spento, livello del volume, ...). Ma per una completa modellizzazione dobbiamo considerare anche alcuni suoi **comportamenti** modificabili mediante i controlli esterni (il telecomando e/o i pulsanti):

- accenditi/spegniti;
- cambia canale;
- alza/abbassa il volume;
- modifica la luminosità;
- ...

Object Oriented Programming

L'interesse e la definitiva diffusione dell'approccio OOP sono dovuti al passaggio, avvenuto nel decennio 1980-1990, dalle precedenti architetture informatiche monopiattaforma a quelle multipiattaforma distribuite.
Questa evoluzione ha comportato alcuni problemi da risolvere, in particolare la questione di come distribuire i dati e le funzioni che costituivano le precedenti applicazioni monolitiche su una pluralità di sistemi cooperanti, secondo quale logica e con quale modalità di colloquio tra diversi moduli applicativi.
L'OOP ha fornito a tali domande risposte innovative e tecnicamente adeguate, in particolare sotto il profilo tecnico-organizzativo nello sviluppo del software.
Un'applicazione a oggetti è costituita da un insieme di moduli software logicamente indipendenti in cui le classi definiscono modelli astratti che incapsulano dati e operazioni sui dati stessi (risolvendo la tradizionale distinzione tra dati e funzionalità) e che interagiscono tra loro tramite scambio di «messaggi».

In pratica, per definire il modello del nostro oggetto, abbiamo effettuato:

- un'astrazione sui dati mediante le sue caratteristiche (attributi);
- un'astrazione funzionale individuando le azioni che può compiere (operazioni).

Queste due astrazioni sono alla base dell'**approccio orientato agli oggetti** (*Object Oriented* o **OO** in breve), sia in relazione alla **progettazione** (**OOD**, *Object Oriented Design*) sia alla **programmazione** (**OOP**, *Object Oriented Programming*).

Il livello di complessità dell'architettura interna di un televisore (vista come insieme dei suoi componenti) non è banale; ciò nonostante, la maggior parte di noi sa come usarlo – almeno nelle funzionalità di base – utilizzando pochi comandi che possono essere attivati mediante un telecomando o i pulsanti presenti al suo esterno. In pratica non interessa (e non deve interessare) sapere com'è fatto dentro per poterlo utilizzare. D'altra parte non è pensabile comandarlo interagendo direttamente con i componenti elettronici che costituiscono la sua struttura interna, usando strumenti come pinze o cacciaviti, per ragioni sia di praticità sia di sicurezza: potremmo danneggiare il televisore o rimanere fulminati da una scarica elettrica!

La precedente riflessione è inerente a uno dei fondamenti dell'approccio OO: il **principio di *information hiding*** (**occultamento dell'informazione**), che stabilisce la netta separazione tra l'implementazione (struttura interna) e l'interfaccia (comandi per l'utilizzo) di un oggetto.

1 Tipi di dato astratto e principio di *information hiding*

Quando un informatico si riferisce ai tipi di dato viene immediatamente da pensare ai tipi elementari come i numeri interi, i numeri decimali, le stringhe di caratteri, spesso utilizzati nel contesto di un programma.

Ma un tipo di dato è caratterizzato da due aspetti fondamentali: un **insieme di valori** e un **insieme di operazioni** che possono essere a esso applicate.

ESEMPIO Per i valori numerici interi compresi in un intervallo finito sono definite le classiche operazioni aritmetiche (somma, sottrazione, moltiplicazione, divisione). Le stringhe di testo sono sequenze finite di caratteri per le quali sono definite operazioni che consentono, per esempio, di determinare o modificare un singolo carattere, di concatenare due stringhe, di estrarre una sottostringa e così via.

OSSERVAZIONE In matematica i valori e le operazioni di uno specifico tipo di dato sono definiti mediante equazioni, assiomi o altre descrizioni

formali. Un'algebra in prima approssimazione viene definita come un insieme di valori associato alle operazioni definite su di esso.

Con l'espressione **tipo di dato astratto** (**ADT**, *Abstract Data Type*) ci si riferisce a un tipo di dato completamente specificato, ma indipendentemente da una sua particolare implementazione. L'implementazione di un ADT è demandata allo sviluppatore del software sia per il contenuto informativo (**dati** o **attributi**), sia per le operazioni specifiche associate al tipo di dato (**metodi**).

Un ADT è caratterizzato dalle seguenti proprietà caratteristiche:

- la definizione di un nuovo tipo di dato;
- la definizione di un insieme di operazioni sul tipo di dato che costituiscono la sua interfaccia;
- le operazioni previste dall'interfaccia sono l'unico meccanismo tramite il quale è possibile interagire con il dato;
- il comportamento delle operazioni è definito mediante condizioni prestabilite.

OSSERVAZIONE Nella programmazione orientata agli oggetti l'elemento centrale è il dato: l'idea alla base dei tipi di dato astratti è infatti quella di definire una struttura di dati con associate opportune operazioni su di essa. Quindi è il tipo di dato che esporta le procedure necessarie alla sua gestione.

ESEMPIO La maggior parte dei linguaggi di programmazione non possiede tra i propri tipi di dato elementari le frazioni. Un ADT che definisca come nuovo tipo una frazione avrà come attributi i valori del numeratore e del denominatore della frazione e come operazioni le usuali operazioni aritmetiche (somma, sottrazione, moltiplicazione, divisione) realizzate mediante gli algoritmi che consentono di ottenere una frazione come risultato di un'operazione che interessa due frazioni come operandi.

Il principio fondamentale di progettazione degli ADT è quello dell'*information hiding* o **incapsulamento** che stabilisce la netta separazione tra l'interfaccia esposta da un ADT all'esterno per essere utilizzato e la sua implementazione interna che deve invece rimanere nascosta. Un obiettivo importante del principio di *information hiding* è rendere invisibili all'esterno di un ADT le scelte implementative che possono essere soggette a modifiche, proteggendo in questo modo il codice che utilizza l'ADT dalle conseguenze dovute all'eventuale cambiamento di tali scelte.

ESEMPIO Il segno di una frazione definita mediante un ADT può essere rappresentato in vari modi: in modo separato dai valori del numeratore e del denominatore (considerati in questo caso sempre positivi), come segno del solo numeratore (considerando in questo caso il denominatore sempre positivo), come combinazione algebrica dei segni del numeratore e del denominatore. L'eventuale variazione della modalità di rappresentazione del segno internamente all'ADT non dovrebbe in nessun modo alterarne l'interfaccia esterna.

L'incapsulamento, inoltre, garantisce un corretto utilizzo del tipo di dato prevenendo eventuali errori nella sua gestione da parte del codice che lo utilizza.

ESEMPIO Un ADT che definisce una frazione deve impedire che il denominatore sia erroneamente impostato al valore zero: l'impossibilità di accedere dall'esterno direttamente ai valori dei dati (attributi) consente allo sviluppatore che implementa le operazioni (metodi) di evitare questa condizione di errore.

Ovviamente è necessario, sia in fase di progettazione sia di implementazione di un ADT, porre molta cura nella definizione della sua interfaccia esterna. L'adeguato funzionamento del codice che usa un ADT è garantito dal rispetto delle specifiche dell'interfaccia e dalla corretta utilizzazione delle operazioni che l'interfaccia stessa rende disponibili.

OSSERVAZIONE Dal momento che la correttezza dell'accesso e della variazione dei valori memorizzati da un ADT sono garantiti dalle operazioni dell'interfaccia e che quest'ultima costituisce l'unico meccanismo di accesso alla rappresentazione interna dei dati, nello sviluppo del software il programmatore può utilizzare gli ADT come componenti «pronti all'uso» preoccupandosi esclusivamente della logica generale dell'applicazione in relazione alle funzionalità che vuole realizzare.

Riguardo alla modularità delle applicazioni possiamo affermare che il principio di *information hiding* è fondamentale nella progettazione e realizzazione di software: la suddivisione in moduli e la corretta specificazione delle loro interfacce ne permettono lo sviluppo e la verifica indipendente, in modo che ogni eventuale errore rimanga relativo a un singolo modulo (a meno che non si tratti di un errore di natura più complessa relativo all'errata modellazione dell'interazione tra moduli).

La correzione di un errore implica quindi la modifica del solo modulo che lo contiene, senza effetti collaterali sugli altri moduli nell'ottica di una filosofia di progetto per cui, come afferma Grady Booch[1], «nessuna parte di un sistema complesso dovrebbe dipendere dai dettagli interni di qualsiasi altra parte».

1. Grady Booch è un noto progettista e metodologo nel campo dell'ingegneria del software *object-oriented*.

Nella pratica della programmazione il principio di *information hiding* viene realizzato ricorrendo a una tecnica che, in generale, prevede la classificazione degli attributi e dei metodi in **pubblici** o **privati**. Un elemento pubblico sarà visibile nell'interfaccia che un oggetto espone verso l'esterno, mentre un elemento privato è confinato all'interno e per l'utente dell'oggetto esso risulterà invisibile.

OSSERVAZIONE Tra i termini *information hiding* e incapsulamento, usati spesso come sinonimi, sussiste una differenza concettuale: il primo è il principio teorico su cui si basa l'altro. In base all'*information hiding* una parte di un programma può nascondere informazioni incapsulandole in un costrutto dotato di interfaccia. L'incapsulamento non è però garanzia di *information hiding*, poiché potrebbe, se mal utilizzato o in casi particolari, non nascondere i dettagli implementativi.

ESEMPIO

Il concetto dell'*information hiding* è rappresentato graficamente in FIGURA 1 dove la struttura privata interna di un oggetto che modella un televisore è incapsulata e protetta dall'interfaccia che espone all'esterno le sole operazioni pubbliche che permettono di interagire con l'oggetto stesso.

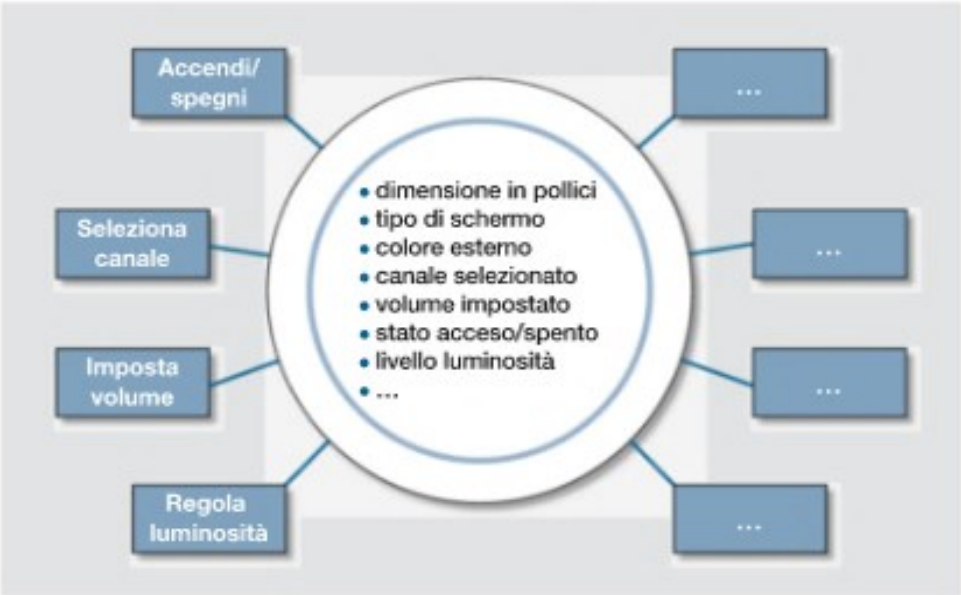


FIGURA 1

Pur essendo possibile definire attributi di tipo pubblico, una buona prassi di progettazione e programmazione sconsiglia questa pratica; lasciando gli attributi privati, essi risultano accessibili esclusivamente mediante i metodi pubblici che l'oggetto espone nella sua interfaccia. Stabilendo, per esempio, che il livello di volume del nostro televisore è impostabile in un intervallo compreso tra i valori 0 e 100, è inopportuno lasciare al codice che utilizza l'oggetto la possibilità di modificare senza alcun controllo tale valore: sarebbe infatti possibile impostare dei valori fuori dell'intervallo definito causando un errore. È preferibile permettere la gestione del livello di volume esclusivamente mediante specifici metodi pubblici – come *aumentaVolume* e *diminuisciVolume* – che nella loro implementazione dovranno impedire che il valore impostato fuoriesca dall'intervallo predefinito.

2 Classi e oggetti, attributi e metodi nei diagrammi UML

Il concetto di **classe** è alla base della progettazione e della programmazione orientate agli oggetti: una classe rappresenta la descrizione di una specifica categoria di oggetti individuati da comportamenti e caratteristiche simili.

ESEMPIO

La classe *Televisore* può essere usata per definire gli oggetti *televisori*.

Come abbiamo già visto i comportamenti (le funzionalità) sono denominati **metodi** o **operazioni**, mentre le componenti informative (i dati) sono denominate **proprietà** o **attributi**.

Nel contesto della OOD/OOP, una **classe** rappresenta un modello formale per la descrizione di un certo tipo di oggetti definendone gli attributi, i metodi e le caratteristiche dell'interfaccia.

Nel codice di un qualsiasi linguaggio di programmazione OO, a partire dalla definizione di una classe, è possibile creare (il termine tecnico è **istanziare**) oggetti simili che condividono lo stesso insieme di attributi, lo stesso insieme di metodi e la stessa interfaccia. Ogni oggetto (**istanza** della classe) è però un'entità autonoma con propri valori per gli attributi.

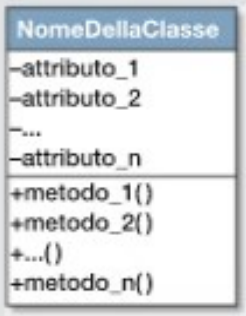
ESEMPIO A partire dalla classe *Televisore* è possibile istanziare due oggetti distinti *televisore_soggiorno* e *televisore_cucina* dove il valore dell'attributo che definisce la dimensione dello schermo in pollici può essere uguale a 32 per *televisore_soggiorno* e a 24 per *televisore_cucina*.

Nella programmazione OO la creazione di un oggetto determina due azioni consequenziali:

- allocare un'area di memoria per la memorizzazione dell'oggetto stesso;
- inizializzare i valori degli attributi che costituiscono la componente informativa dell'oggetto.

La seconda azione viene espletata dal **costruttore** della classe, ossia da uno speciale metodo, normalmente denominato con lo stesso nome della classe.

Nella progettazione OO, tra le varie proposte che si sono succedute nel tempo per la definizione di un formalismo che permettesse la rappresentazione delle classi, **UML** (*Unified Modeling Language*) è sicuramente quella che ha riscosso il maggior successo. Tra i vari formalismi grafici che UML mette a disposizione per descrivere i diversi aspetti di un sistema software a molteplici livelli di dettaglio, faremo riferimento al **diagramma delle classi** (*class diagram*) che, come è deducibile dal nome, fornisce una notazione grafica per formalizzare le classi e le relazioni che intercorrono tra di esse. Definire una classe utilizzando un *class diagram* UML è piuttosto semplice; lo schema generale del diagramma relativo a una singola classe è infatti il seguente



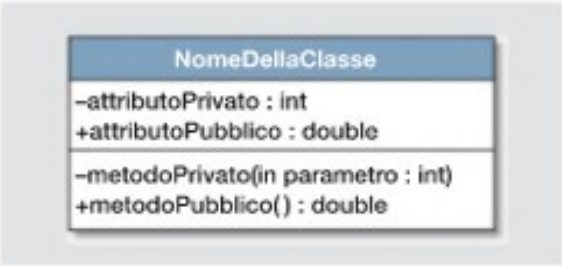
Unified Modeling Language

L'acronimo UML (*Unified Modeling Language*) identifica, nell'ambito dell'ingegneria del software, un insieme di linguaggi grafici di modellazione basato sul paradigma OO.
Esso fu proposto nel 1996 da Grady Booch, James Rumbaugh e Ivar Jacobson (universalmente conosciuti come *los tres amigos*) con la supervisione dello *Object Management Group*, il consorzio che tuttora ne gestisce lo standard.
UML nacque con l'intenzione di unificare gli approcci precedenti, raccogliendone gli aspetti migliori per la definizione di uno standard industriale unificato che comprendesse diversi formalismi grafici, tra cui: il diagramma delle classi, il diagramma degli oggetti, il diagramma dei casi d'uso, il diagramma di sequenza, il diagramma degli stati, il diagramma delle attività, il diagramma delle comunicazioni, il diagramma di *deployment*.
Essendo un set di linguaggi grafici, l'UML svolge, nella comunità della programmazione a oggetti, un'importante funzione per la progettazione del software, indipendentemente dall'effettivo linguaggio di programmazione utilizzato.
È ormai uso comune utilizzare i diagrammi UML per descrivere modelli di analisi e soluzioni progettuali in modo sintetico, comprensibile e rigoroso.
L'ultima versione del linguaggio, la 2.0, è stata consolidata nel 2004 e ufficializzata da OMG nel 2005.

dove:

- nella prima sezione a partire dall'alto si inserisce il nome della classe;
- la seconda sezione è relativa alla definizione degli attributi (proprietà);
- la terza e ultima sezione è relativa alla definizione delle operazioni (metodi).

È inoltre possibile classificare le componenti private e quelle pubbliche premettendo ai singoli nomi rispettivamente i simboli «–» e «+»:



OSSERVAZIONE Si noti che per ogni metodo, oltre al suo livello di visibilità (pubblica o privata) è necessario definire:

- i nomi e i tipi degli eventuali parametri e il loro ruolo (in/out/in-out);
- il tipo dell'eventuale valore restituito.

ESEMPIO

Il diagramma UML di FIGURA 2 definisce la classe *Televisore*.

FIGURA 2

Classi parametriche

Una *classe parametrica* è una classe definita utilizzando dei tipi generici che dovranno essere specificati come tipi reali da parte del codice che ne istanzierà gli oggetti: in questo modo è possibile generalizzare la funzionalità di una classe.
Per esempio, una classe che implementa una lista di elementi può essere facilmente definita in modo indipendente dal tipo di elementi che vi potranno essere memorizzati: il tipo specifico degli elementi potrà essere indicato al momento della creazione di una lista e la stessa classe consentirà di gestire liste di elementi di tipo diverso.

In relazione a questo esempio si noti che:

- tutti gli attributi sono privati e tutti i metodi sono pubblici;
- per accedere agli attributi privati sono stati definiti metodi convenzionalmente noti come *getter/setter*: un metodo il cui nome inizia con il prefisso *get*, seguito dal nome di un attributo della classe, restituisce il valore dell'attributo; un metodo il cui nome inizia con il prefisso *set*, seguito dal nome di un attributo, ha lo scopo di impostare un nuovo valore la cui congruità viene controllata dal metodo stesso;
- non per tutti gli attributi sono stati definiti dei metodi *setter*: in particolare alcuni attributi di un televisore non sono modificabili nel corso della sua esistenza e possono essere impostati solo dal metodo costruttore che assume il nome della classe stessa;
- i metodi che consentono di modificare gli attributi aumentandone o diminuendone il valore devono essere implementati in modo che non possano impostare valori inferiori al minimo valore predefinito, o superiori al massimo valore predefinito per ogni attributo.

ESEMPIO

Il diagramma UML di FIGURA 3 definisce la classe *Frazione*.
A proposito di questo ulteriore esempio si può osservare che:

- i due metodi *MCD* ed *mcm*, che calcolano rispettivamente il massimo comun divisore e il minimo comune multiplo tra due numeri interi *a* e *b* sono entrambi privati e quindi usati solo internamente alla classe; essi sono infatti funzionali alle operazioni di semplificazione di una frazione (*MCD*) ed (*mcm*) e al calcolo della somma/sottrazione tra due frazioni;
- i metodi relativi alla semplificazione, al calcolo dell'opposta e della reciproca di una frazione in aggiunta a quelli di somma, sottrazione, moltiplicazione e divisione restituiscono come risultato un oggetto di tipo *Frazione*;
- i metodi relativi alle quattro operazioni algebriche sono applicabili a coppie di oggetti di tipo *Frazione* (per esempio, disponendo dei due oggetti di tipo *Frazione* *f*1 ed *f*2 la somma di questi, che è anch'essa una frazione, potrebbe essere invocata con una sintassi del tipo *f1.somma(f2)* dove *f*1 è l'oggetto a cui viene applicato il metodo somma mentre *f*2 è l'argomento di tale metodo).

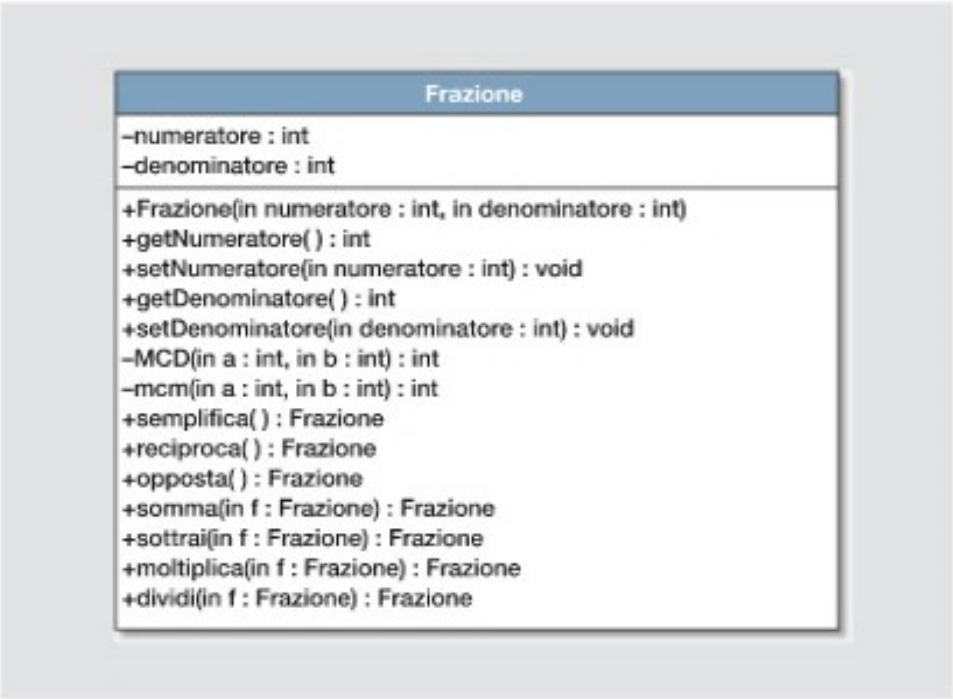


FIGURA 3

3 Interazione tra oggetti e diagrammi UML di sequenza

In un sistema software gli oggetti interagiscono tra di loro scambiandosi **messaggi**: un messaggio è un'informazione che viene scambiata tra due oggetti.

> In un linguaggio di programmazione OO lo scambio di messaggi avviene mediante invocazione dei metodi degli oggetti istanziati a partire dalle classi definite.

Il messaggio può essere:

- **sincrono**: l'emittente rimane in attesa di una risposta;
- **asincrono**: l'emittente non rimane in attesa di una risposta che può eventualmente essere ricevuta in un momento successivo.

Un messaggio generato in risposta a un precedente messaggio, e al quale eventualmente fa riferimento anche in relazione al suo contenuto informativo, è detto **messaggio di risposta**.

È possibile rappresentare graficamente lo scambio di messaggi tra oggetti utilizzando un **diagramma UML di sequenza** (*sequence diagram*). Un diagramma di sequenza è un tipo di diagramma dello standard UML ideato per descrivere una determinata sequenza di interazioni.

ESEMPIO

La FIGURA 4 rappresenta il diagramma di sequenza che descrive l'acquisto di un oggetto in un negozio da parte di un cliente.

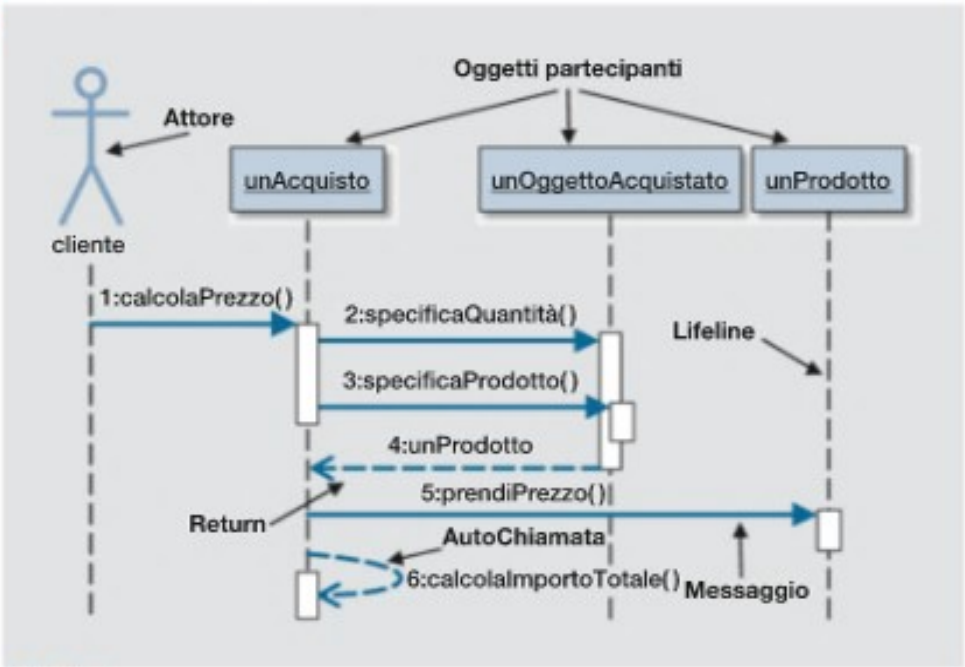


FIGURA 4

In un diagramma di sequenza si distinguono i seguenti elementi.

- **Istanze di classi (oggetti)**: sono rappresentate da rettangoli riportanti il nome della classe e l'identificatore dell'oggetto (con la notazione: ***og-***

Design pattern

I *design pattern* sono soluzioni generali predefinite di progettazione applicabili a problemi ricorrenti nella progettazione del software in contesti eterogenei. Questo tipo di approccio è stato proposto da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, ormai universalmente noti a tutti gli informatici come la *Gang of Four* o mediante l'acronimo collettivo GoF.
L'idea di base della proposta è piuttosto semplice: dal momento che la qualità della progettazione software deriva dall'esperienza del progettista – un progettista esperto nel risolvere nuovi problemi anche in nuovi contesti utilizza comunque schemi di soluzioni già sperimentate che hanno fornito buoni risultati in precedenza – è possibile individuare, formalizzare (spesso utilizzando diagrammi UML) e pubblicare le soluzioni progettuali che risolvono i problemi che si presentano più frequentemente in modo indipendente dal contesto.
In questo modo si mettono a disposizione della comunità degli sviluppatori software privi della necessaria esperienza soluzioni predefinite di qualità almeno per i problemi progettuali comuni.

3 Interazione tra oggetti e diagrammi UML di sequenza

In un sistema software gli oggetti interagiscono tra di loro scambiandosi **messaggi**: un messaggio è un'informazione che viene scambiata tra due oggetti.

> In un linguaggio di programmazione OO lo scambio di messaggi avviene mediante invocazione dei metodi degli oggetti istanziati a partire dalle classi definite.

Il messaggio può essere:

- **sincrono**: l'emittente rimane in attesa di una risposta;
- **asincrono**: l'emittente non rimane in attesa di una risposta che può eventualmente essere ricevuta in un momento successivo.

Un messaggio generato in risposta a un precedente messaggio, e al quale eventualmente fa riferimento anche in relazione al suo contenuto informativo, è detto **messaggio di risposta**.

È possibile rappresentare graficamente lo scambio di messaggi tra oggetti utilizzando un **diagramma UML di sequenza** (*sequence diagram*). Un diagramma di sequenza è un tipo di diagramma dello standard UML ideato per descrivere una determinata sequenza di interazioni.

ESEMPIO

La FIGURA 4 rappresenta il diagramma di sequenza che descrive l'acquisto di un oggetto in un negozio da parte di un cliente.

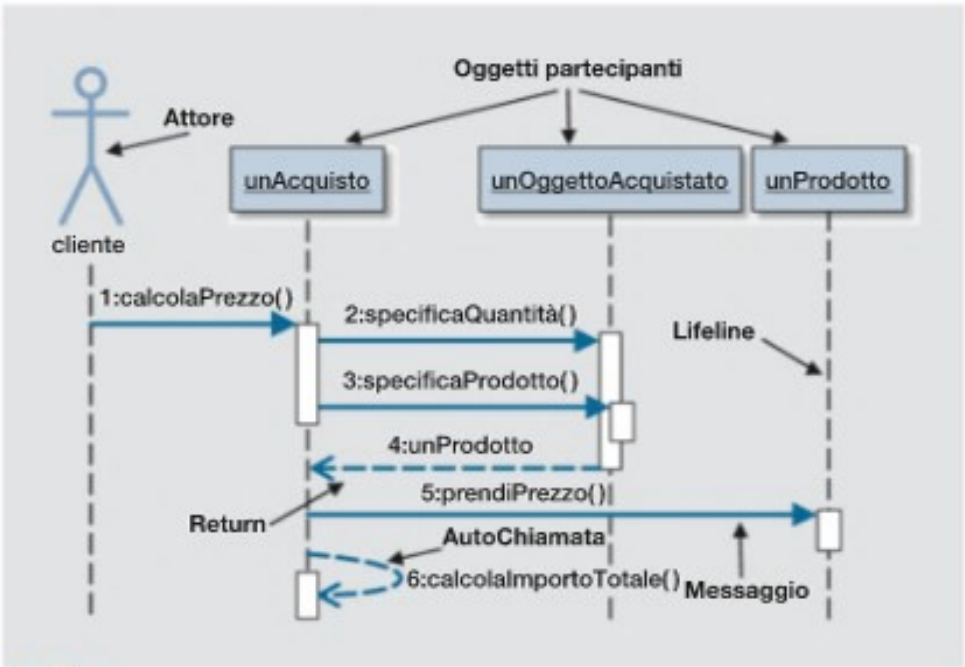


FIGURA 4

In un diagramma di sequenza si distinguono i seguenti elementi.

- **Istanze di classi (oggetti)**: sono rappresentate da rettangoli riportanti il nome della classe e l'identificatore dell'oggetto (con la notazione: ***og-***

Design pattern

I *design pattern* sono soluzioni generali predefinite di progettazione applicabili a problemi ricorrenti nella progettazione del software in contesti eterogenei. Questo tipo di approccio è stato proposto da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, ormai universalmente noti a tutti gli informatici come la *Gang of Four* o mediante l'acronimo collettivo GoF.
L'idea di base della proposta è piuttosto semplice: dal momento che la qualità della progettazione software deriva dall'esperienza del progettista – un progettista esperto nel risolvere nuovi problemi anche in nuovi contesti utilizza comunque schemi di soluzioni già sperimentate che hanno fornito buoni risultati in precedenza – è possibile individuare, formalizzare (spesso utilizzando diagrammi UML) e pubblicare le soluzioni progettuali che risolvono i problemi che si presentano più frequentemente in modo indipendente dal contesto.
In questo modo si mettono a disposizione della comunità degli sviluppatori software privi della necessaria esperienza soluzioni predefinite di qualità almeno per i problemi progettuali comuni.

getto : classe), o più semplicemente un nome dal quale si desume che si tratta di un'istanza della classe (*unAcquisto*, *unProdotto*, ...).

- **Attori**: se presenti, sono riportati sulla sinistra con le frecce di interazione rivolte verso gli oggetti del sistema.
- **Messaggi**: sono rappresentati come frecce da un attore a un oggetto, o tra due oggetti; i messaggi di ritorno, se riportati, hanno la linea della freccia tratteggiata. I messaggi sincroni terminano con una freccia triangolare piena, mentre i messaggi asincroni terminano con una freccia aperta semplice. Un messaggio può anche insistere sullo stesso oggetto che lo invia. L'ordine dei messaggi – dall'alto verso il basso – rispetta la sequenza temporale con cui sono scambiati; per maggiore chiarezza essi possono essere preceduti da un numero che ne indica l'esatta successione.
- **Linee di vita (*lifeline*)**: sono linee tratteggiate verticali che partono dal rettangolo rappresentativo dell'oggetto; indicano il periodo temporale di vita dell'oggetto, dalla sua costruzione alla sua distruzione.
- **Barre di attivazione (*activation box*)**: sono rappresentate da un rettangolo sovrapposto alla linea di vita di un oggetto che riceve un messaggio: rappresentano convenzionalmente il tempo di elaborazione della richiesta giunta all'oggetto.

I diagrammi UML di sequenza sono utilizzati in diverse fasi del ciclo di vita di sviluppo di un'applicazione software.

- Nella fase di analisi un *sequence diagram* può rappresentare graficamente uno scenario di un caso d'uso: in questo caso il ruolo degli oggetti sarà recitato da un generico oggetto denominato **sistema**, saranno presenti altri oggetti (indicanti, per esempio, alcuni componenti architetturali quali *server*, *database*, ...) solo se rappresentano elementi vincolati dall'analisi.
- Nella prima parte della fase di progettazione i *sequence diagram* descrivono scenari di casi d'uso in cui come oggetti compaiono istanze delle classi individuate e come messaggi le operazioni riportate nel diagramma delle classi.
- Nella successiva progettazione di dettaglio i *sequence diagram* descrivono realizzazioni di metodi complessi, dove come oggetti compaiono istanze delle classi del diagramma delle classi del progetto di dettaglio e come messaggi le invocazioni dei metodi degli oggetti. In questo caso al posto degli elementi architetturali devono comparire oggetti istanza delle classi che ne consentono l'accesso (per esempio, invece che *database* dovrebbe comparire l'oggetto che consente di interagire con il database).

OSSERVAZIONE Per la descrizione di algoritmi, i *sequence diagram* – che sono stati specificatamente ideati per descrivere le interazioni – non sono lo strumento adatto: UML dispone allo scopo dei formalismi grafici *activity diagram* e *state-chart diagram*.

ESEMPIO

Un sistema distributore automatico di bevande in bottiglia può essere verosimilmente scomposto in quattro oggetti di base:

- il **cruscotto esterno**, ovvero l'interfaccia che la macchina presenta all'utente;
- il **ricevitore delle monete**, cioè il componente in cui vengono introdotte e raccolte le monete;
- il **controllo**, ovvero il dispositivo che gestisce l'intera macchina;
- il **contenitore delle bottiglie**, il sottosistema che contiene i prodotti acquistabili dagli utenti.

Nel diagramma di sequenza possiamo individuare le seguenti azioni (FIGURA **5**):

- il cliente inserisce le monete nella macchina;
- il cliente esegue la selezione del prodotto desiderato;
- le monete vengono acquisite dal ricevitore;
- il dispositivo di controllo verifica se il prodotto richiesto è disponibile;
- il ricevitore delle monete aggiorna il suo contenuto;
- il dispositivo di controllo abilita il contenitore delle bottiglie all'erogazione del prodotto richiesto.

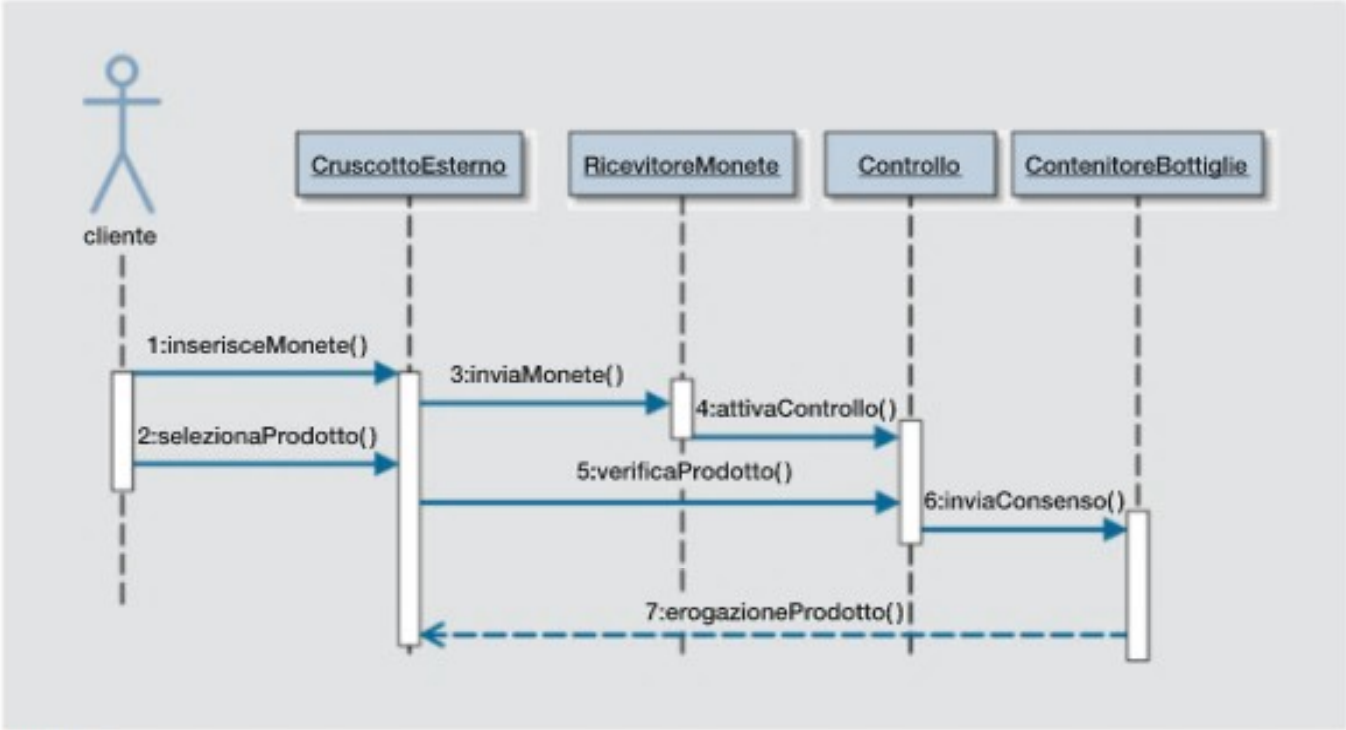


FIGURA **5**

ESEMPIO

Un sistema che rappresenta un televisore con un servizio *pay-per-view* può essere rappresentato da tre oggetti:

- il **televisore**, cioè l'interfaccia tramite la quale opera l'utente (ovviamente dotato di dispositivo *pay-per-view* per l'introduzione della scheda di pagamento);
- il **controllo remoto**, che effettua il controllo della scheda di pagamento e del suo credito, da cui verrà detratto il costo del programma prescelto;
- il **fornitore del servizio**, cioè il sottosistema che memorizza i programmi televisivi che vengono acquistati dall'utente.

Nel diagramma di sequenza possiamo individuare le seguenti azioni (FIGURA **6**):

- l'utente inserisce la scheda prepagata nel televisore;
- l'utente esegue la selezione del programma desiderato;
- il controllo remoto verifica se il credito è sufficiente per pagare il servizio richiesto;
- il credito della scheda viene diminuito del costo previsto per la fruizione del programma;
- il dispositivo di controllo informa il sistema di erogazione del servizio che può rendere disponibile il programma prescelto.

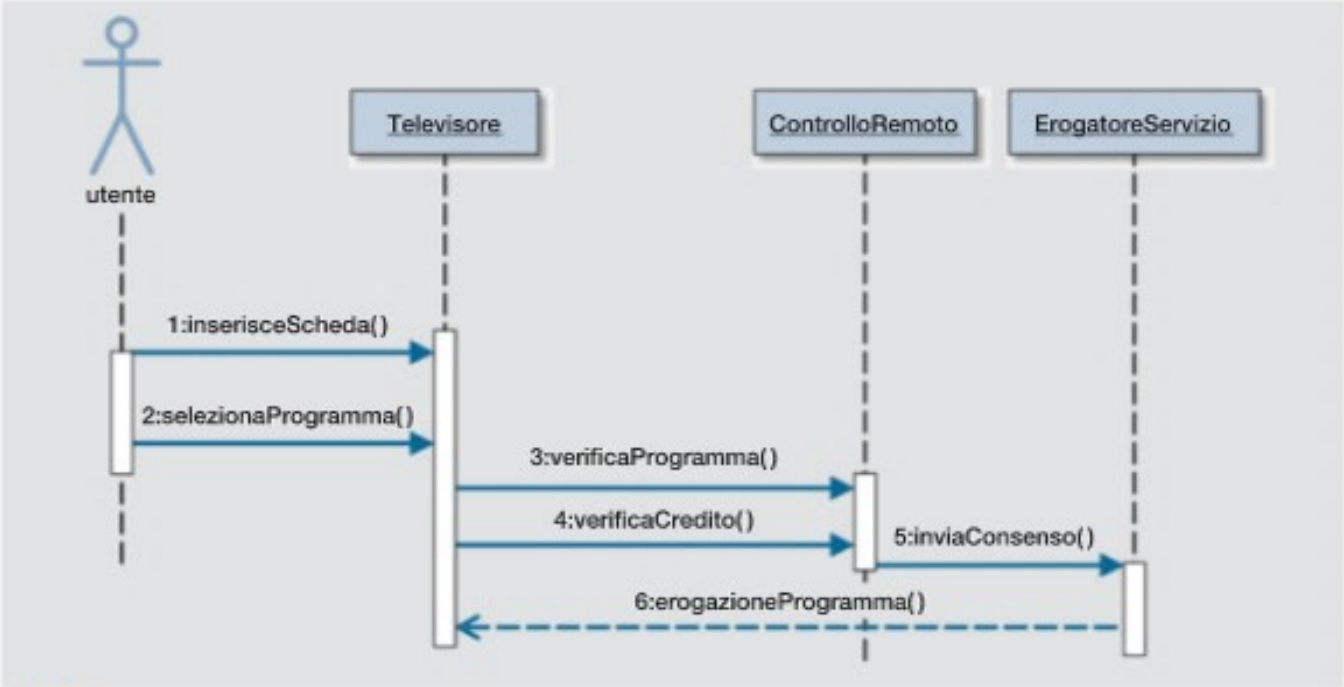


FIGURA 6

4 Ereditarietà e polimorfismo

Un concetto fondamentale dell'OOD/OOP è quello di **ereditarietà**: utilizzando questo meccanismo è infatti possibile in un linguaggio OO creare nuove classi a partire da classi già esistenti ereditandone le caratteristiche (attributi e/o metodi), aggiungendone di nuove o ridefinendone alcune. L'ereditarietà è finalizzata alla creazione di gerarchie di classi e grazie a essa si estende la possibilità di riutilizzare componenti comuni a più classi della stessa gerarchia.

Per mezzo dell'ereditarietà tra classi è possibile definire una classe generale (**classe base** o **superclasse**) avente la funzione di definire le caratteristiche comuni a uno specifico insieme di oggetti. Le caratteristiche della classe generale potranno quindi essere ereditate o estese da altre classi (**classi derivate** o **sottoclassi**) che integreranno le caratteristiche della classe base con elementi specifici. Ma le classi derivate, a loro volta, potranno configurarsi come superclassi per nuove classi, realizzando in questo modo la gerarchia.

Una classe derivata da una classe base mantiene le proprietà (attributi) e le operazioni (metodi) della classe base da cui eredita e a questi aggiunge i propri attributi e i propri metodi specifici.

Il meccanismo dell'ereditarietà è particolarmente vantaggioso in un contesto dove si intende riutilizzare in ambiti diversi il codice già sviluppato: per mezzo di esso, infatti, è possibile definire, a partire da una classe data, nuove classi, specificando semplicemente le differenze tra queste e la classe di partenza.

ESEMPIO Avendo definito la classe *Veicoli_a_Motore*, essa sarà una classe base per le classi derivate *Auto* e *Motocicli*, che erediteranno da essa proprietà e operazioni. Pur con le proprie particolarità, infatti, sia le automobili sia i motocicli sono comunque entrambi dei veicoli a motore, in quanto dotati di caratteristiche comuni come la cilindrata, la targa, il numero di telaio, ...). Procedendo nella costruzione della gerarchia, le due sottoclassi potranno a loro volta essere superclassi per altri tipi di automobili (*Tradizionali, Cabriolet, Suv*, ...) o di motocicli (*Moto_Stradali, Moto_Fuoristrada*, ...).

In un diagramma UML delle classi l'ereditarietà – che in questo contesto assume la denominazione di **generalizzazione** – viene rappresentata in modo intuitivo mediante frecce che collegano le classi derivate alle classi base (le estremità delle frecce sono triangolari, ma vuote).

ESEMPIO Il diagramma di FIGURA 7 illustra la gerarchia delle classi ipotizzata nell'esempio precedente.

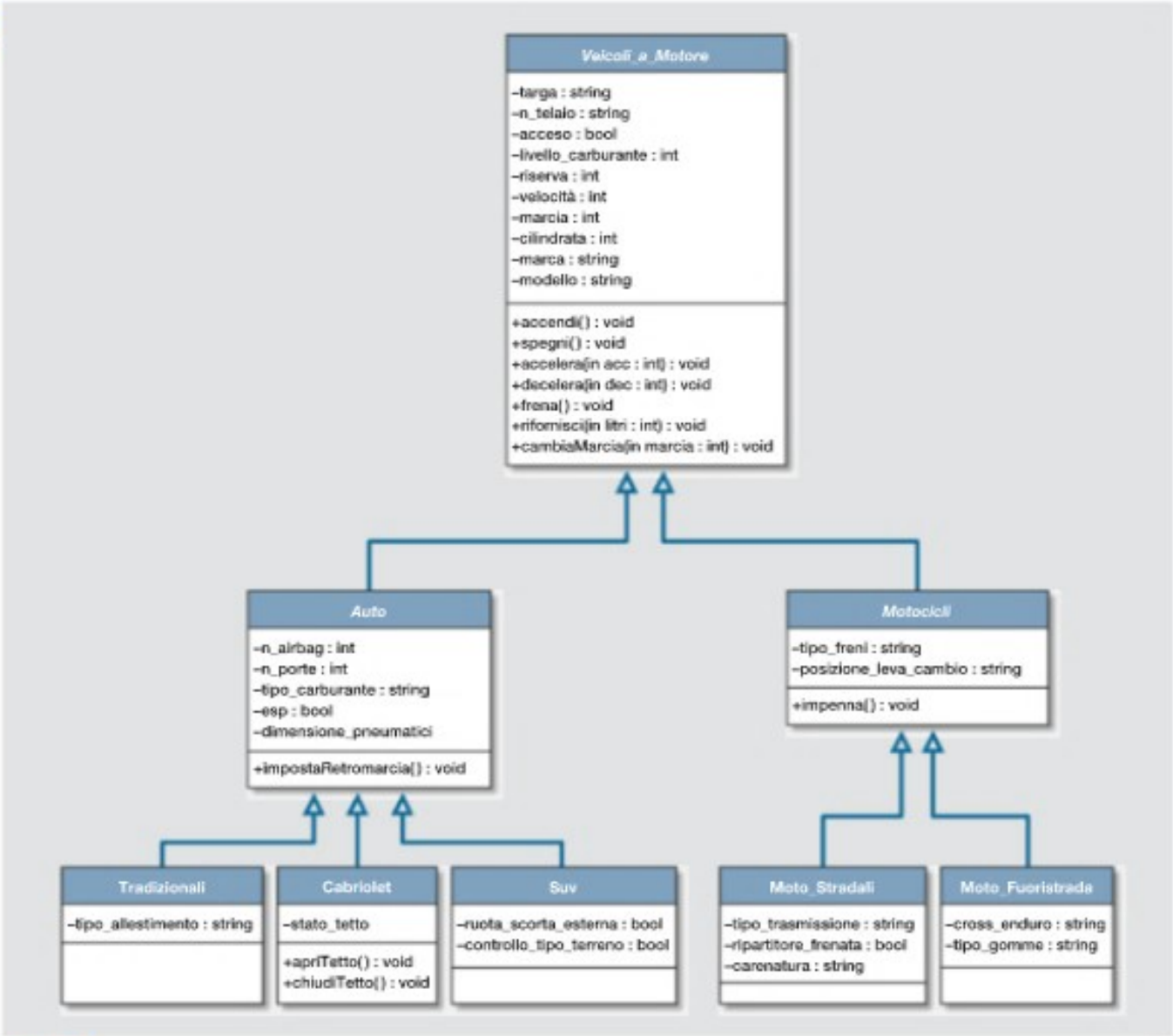


FIGURA 7

Il tipo di ereditarietà presentato è noto come **ereditarietà singola**: in questo caso una sottoclasse può derivare al più da una superclasse. Esistono però situazioni in cui una sottoclasse dovrebbe derivare da due o più su-

perclassi: in questo caso il tipo di ereditarietà è definito **ereditarietà multipla**.

ESEMPIO

Dovendo progettare una gerarchia di classi per rappresentare i veicoli in funzione dell'ambiente in cui questi sono capaci di spostarsi, il diagramma UML potrebbe essere strutturato nel modo illustrato in FIGURA **8**.
In questo caso i veicoli anfibi ereditano sia le caratteristiche dei veicoli terrestri sia quelle dei veicoli acquatici.

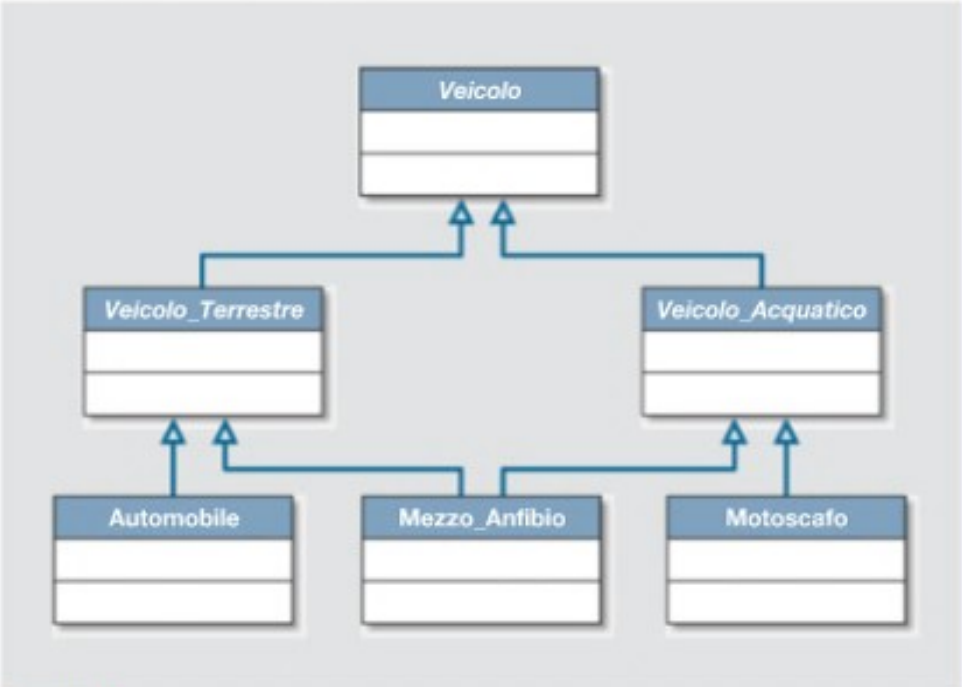


FIGURA **8**

OSSERVAZIONE Tutti i linguaggi di programmazione orientati agli oggetti supportano l'ereditarietà singola, ma solo alcuni consentono di realizzare l'eredità multipla (C++ è tra questi, ma Java, per esempio, la consente solo in una forma molto limitata tramite le cosiddette «interfacce»).

4.1 Polimorfismo

Il concetto di **polimorfismo** degli oggetti, tipico della programmazione OO, è strettamente collegato all'ereditarietà tra classi. Utilizzando il polimorfismo è possibile ottenere comportamenti e risultati diversi invocando gli stessi metodi a carico di oggetti diversi.

Il polimorfismo è fondato sulla possibilità di ridefinire nelle classi derivate i comportamenti originali dei metodi ereditati dalla classe base; in questo modo è possibile ottenere metodi in grado di operare in modo appropriato su oggetti di diversa tipologia: sarà compito dell'ambiente di esecuzione decidere di volta in volta quale sia il metodo corretto da utilizzare sulla base della specifica classe di cui l'oggetto è istanza.

Mediante il polimorfismo è possibile trattare gli oggetti istanza delle classi di una gerarchia derivata da una classe base come se fossero istanze della classe base stessa.

ESEMPIO

Dovendo scrivere del codice in un linguaggio di programmazione orientato agli oggetti per calcolare l'area di varie figure geometriche, è sensato definire una classe generale *Figura_Geometrica* dalla quale derivare le classi che rappresentano le particolari figure geometriche (*Quadrato, Rettangolo, Trapezio*, ...). Il diagramma UML delle classi è quindi quello di FIGURA 9.

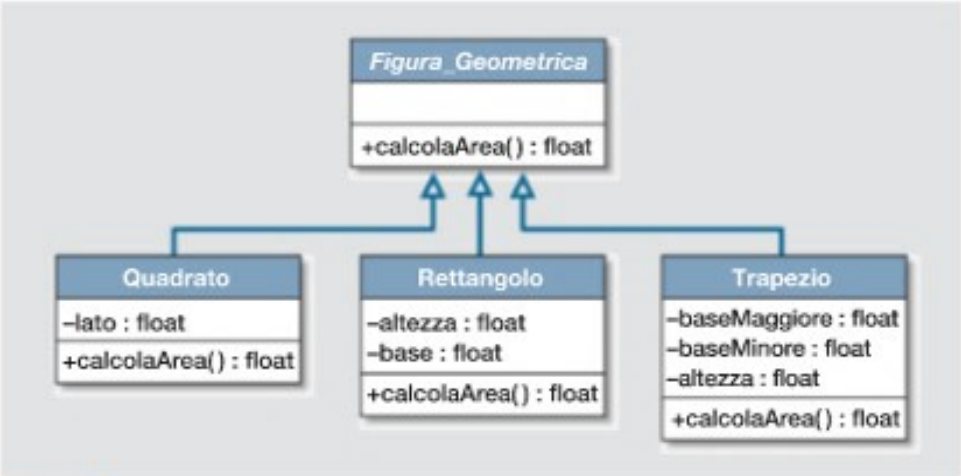


FIGURA 9

Il metodo *calcolaArea* della classe *Figura_Geometrica* è ridefinito in tutte le classi derivate: questo significa che ogni classe implementa uno specifico algoritmo di calcolo dell'area mantenendo solo la firma del metodo ereditato. Per calcolare l'area di una figura geometrica disponendo di un oggetto *figura* istanza di una qualsiasi classe della gerarchia è quindi sufficiente invocare questo metodo:

```
...
area = figura.calcolaArea();
...
```

Applicando il polimorfismo il codice è in grado di determinare il metodo da invocare (quello definito nella classe *Quadrato*, oppure nella classe *Rettangolo*, o nella classe *Trapezio*) in funzione della classe di cui l'oggetto *figura* è istanza. Utilizzando un linguaggio di programmazione non orientato agli oggetti, e quindi privo del polimorfismo, il codice sarebbe invece risultato simile al seguente (l'operatore `isA` determina in questo caso se un oggetto è o meno istanza di una classe):

```
...
if (figura isA Quadrato)
  area = figura.calcolaAreaQuadrato();
else
  {
    if (figura isA Rettangolo)
      area = figura.calcolaAreaRettangolo();
    else
      area = figura.calcolaAreaTrapezio();
  }
...
```

Il confronto evidenzia tutta la potenza del polimorfismo, che consente a oggetti istanza di sottoclassi diverse, derivate dalla stessa classe base, di avere la flessibilità di rispondere in modo differenziato allo stesso tipo di messaggio.

Il maggiore beneficio del polimorfismo è dato dalla conseguente facilità di manutenzione del codice. In riferimento all'esempio precedente,

volendo aggiungere la figura geometrica triangolo, è sufficiente definire la nuova classe *Triangolo* derivandola da *Figura_Geometrica*, ma – se il metodo *calcolaArea* viene correttamente ridefinito nella classe *Triangolo* – non è in alcun modo necessario modificare il codice che lo invoca. Non disponendo invece del polimorfismo, per ogni singola invocazione del metodo di calcolo dell'area di una qualsiasi figura geometrica dovrebbe essere aggiunta un'ulteriore selezione per invocare il metodo corretto *calcolaAreaTriangolo*.

4.2 Metodi astratti e classi astratte

Il funzionamento dell'ereditarietà e del polimorfismo nei linguaggi di programmazione OO prevede esplicitamente la ridefinizione nelle sottoclassi derivate e i metodi definiti in una superclasse; in alcuni casi l'implementazione di un metodo in una classe base diviene puramente formale.

ESEMPIO Il metodo *calcolaArea* della classe *Figura_Geometrica* dell'esempio precedente non è in grado di calcolare l'area di nessuna figura in particolare e probabilmente dovrà limitarsi a restituire un valore fittizio o nullo.

Una classe base può limitarsi a definire la *firma* di alcuni metodi (l'**interfaccia**) tralasciandone l'implementazione che riserva alle classi derivate. Metodi con questa caratteristica prendono il nome di **metodi astratti**. Una classe in cui uno o più metodi sono astratti **fattorizza** le operazioni comuni a tutte le sottoclassi dichiarandone i metodi senza implementarli; a partire da una classe di questo tipo non è possibile istanziare oggetti, perché risulterebbero indefiniti rispetto all'eventuale invocazione dei metodi astratti. Classi che presentano uno o più metodi astratti sono dette **classi astratte**: una classe astratta non viene definita allo scopo di istanziare oggetti, ma esclusivamente per derivarne sottoclassi che specificheranno (implementandole) le operazioni che essa dichiara.

Nei diagrammi UML una classe astratta viene indicata scrivendone la denominazione in carattere corsivo.

ESEMPIO Sviluppando l'esempio relativo alla gerarchia dei veicoli a motore, possiamo affermare che:

- ogni veicolo ha un particolare tipo di propulsore;
- ogni veicolo si sposta in una specifica modalità;
- ogni veicolo si muove in un determinato ambiente (terraferma, acqua, aria, ecc.).

Possiamo quindi affermare che non esiste un veicolo astratto generale, ma solo veicoli reali concreti; per esempio:

- l'automobile che ha un motore a scoppio, si sposta sulle ruote e si muove sulla terraferma;
- la nave che ha un motore marino, si sposta navigando e si muove sull'acqua;
- l'aereo che ha un motore a reazione, si sposta volando e si muove nell'aria.

▶

Interfacce

Alcuni linguaggi di programmazione orientati agli oggetti consentono di definire, oltre alle classi, anche le interfacce: un'interfaccia è analoga a una classe astratta, ma costituisce una pura specifica formale dei comportamenti, limitandosi a dichiarare i propri metodi senza implementarli (tutti i metodi di un'interfaccia sono quindi implicitamente astratti); inoltre non può prevedere attributi che non siano costanti.

Le interfacce possono essere estese – il termine tecnico è **implementate** – per ereditarietà, esattamente come le classi, ma possono essere generalmente strutturate in gerarchie di ereditarietà multipla anche nei linguaggi di programmazione, come Java, in cui questo meccanismo non è disponibile per le classi.

L'introduzione di una classe base astratta *Veicolo* permette di fattorizzare – definire cioè una sola volta per tutte le classi derivate – gli aspetti comuni a tutti gli oggetti.
Nel caso specifico possiamo stabilire che, per descrivere un veicolo reale, è in ogni caso necessario definirne il tipo di propulsore, la modalità di spostamento e l'ambiente in cui si muove. Per ogni specifico tipo di veicolo sarà possibile aggiungere le proprietà caratteristiche, ma in ogni caso non potremo prescindere dal definire almeno i tre aspetti fattorizzati per un veicolo generico (FIGURA **10**).

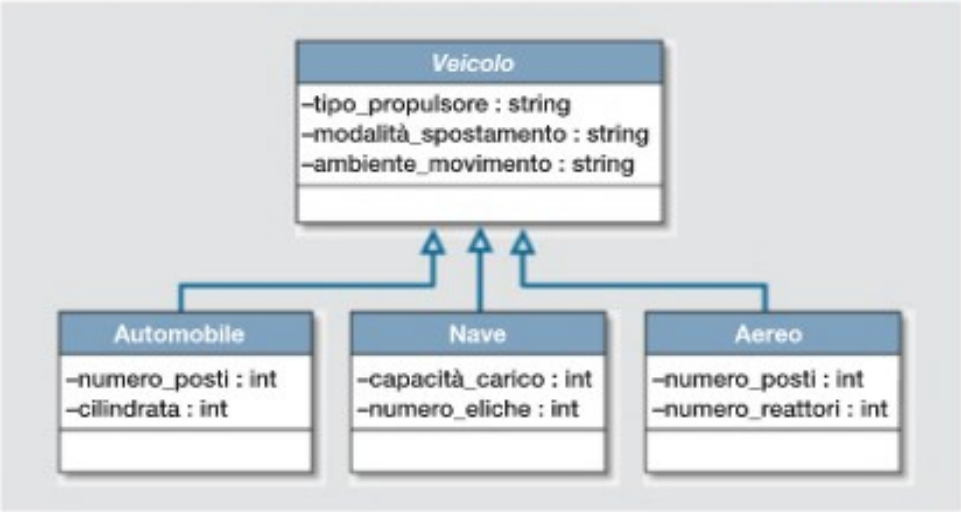


FIGURA **10**

5 Associazioni tra classi

Uno dei punti fondamentali della progettazione software OO consiste nel determinare come gli oggetti creati a partire dalle classi definite collaborino tra di loro. Nella programmazione OO gli oggetti interagiscono tra loro scambiandosi i messaggi che richiedono l'esecuzione di specifici metodi. Oltre al fatto che gli attributi di una classe possono essere del tipo definito da un'altra classe (**associazione statica**), sono gli scambi di messaggi che veicolano con i loro parametri le relazioni tra classi (**associazioni dinamiche**) la cui classificazione e documentazione è fondamentale nella fase di progettazione del software.

Pur esistendo la possibilità di rappresentare associazioni multiple tra classi, il caso più frequente è quello delle associazioni che sussistono tra coppie di classi (**associazioni binarie**) di cui prendiamo in esame le tipologie più significative:

- *dipendenza*;
- *generalizzazione*;
- *composizione*;
- *aggregazione*.

OSSERVAZIONE Nelle associazioni binarie tra classi si definisce **molteplicità** (o **cardinalità**) dell'associazione il numero di oggetti che per ogni classe partecipano all'associazione stessa. I valori possibili sono:

- molti (*);
- uno (1);

- zero o più (0..*);
- zero o più, fino al massimo di N (0..N);
- uno o più (1..*);
- uno o più, fino al massimo di N (1..N);
- da N a M (N..M).

5.1 Dipendenza

Nei diagrammi UML delle classi la **dipendenza**, o **associazione d'uso**, è un tipo di associazione in cui le eventuali variazioni a un elemento (il **fornitore**) possono avere influenza sull'altro elemento (il **cliente**). Questo accade, per esempio, se una classe ha come tipo dei propri attributi un'altra classe, oppure se un metodo di una classe ha parametri del tipo definito da un'altra classe. La simbologia UML adottata per questo tipo di associazione è quella di una freccia tratteggiata orientata dalla classe *cliente* a quella *fornitore*.

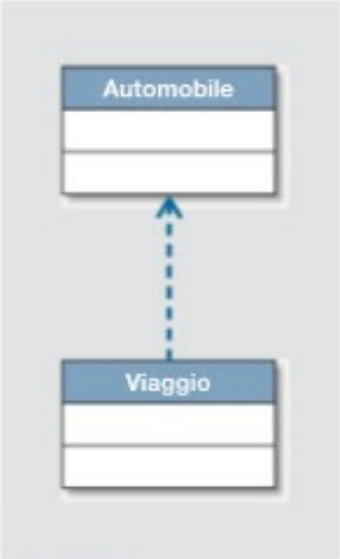


FIGURA **11**

ESEMPIO Un'ipotetica classe *Viaggio* può dipendere dalla classe *Automobile*, se una delle sue operazioni ha un parametro di tipo *Automobile*. Un'associazione di questo tipo si potrebbe interpretare come «*Viaggio* **utilizza** *Automobile*», oppure «*Viaggio* **dipende da** *Automobile*»: la classe *Viaggio* è l'entità cliente, mentre la classe *Automobile* è l'entità fornitore.
Nel diagramma UML delle classi l'associazione di dipendenza è una freccia tratteggiata che parte dalla classe *Viaggio* e arriva alla classe *Automobile* (FIGURA **11**).

OSSERVAZIONE L'associazione di dipendenza dell'esempio precedente indica che un'eventuale modifica della classe *Automobile* potrebbe comportare una modifica della classe *Viaggio*. Alcuni tipi di modifiche alla classe *Automobile*, però, non comporteranno modifiche alla classe *Viaggio*, in particolare quelle relative alla parte privata, oppure quelle relative all'aggiunta di nuove operazioni. Come sempre un attento ricorso al principio di *information hiding* consente di limitare i vincoli fornendo un'ampia possibilità di modifica dell'implementazione privata se l'interfaccia pubblica viene mantenuta inalterata.

5.2 Generalizzazione

L'ereditarietà tra classi realizzata dai linguaggi di programmazione OO è un'associazione comune e nel contesto dei diagrammi UML è definita **generalizzazione**. La rappresentazione UML della generalizzazione consiste in una freccia con estremità triangolare che unisce la classe derivata alla classe base.

L'associazione tra classi in cui una generalizza l'altra è nota come relazione ***is-A***: un oggetto istanza di una sottoclasse è infatti, grazie al polimorfismo, anche istanza della relativa superclasse.

ESEMPIO Il diagramma UML di FIGURA **12** rappresenta la classe *Veicolo_ Terrestre* come generalizzazione della classe *Automobile* (la classe *Automobile* è di conseguenza una «specializzazione» della classe *Veicolo_ Terrestre*).

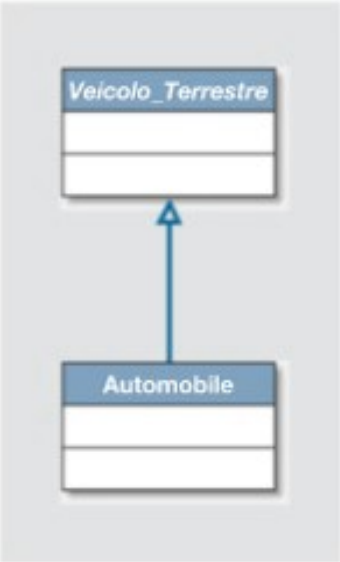


FIGURA **12**

5.3 Composizione

Un tipo di associazione interessante è la **composizione**. Essa è un'associazione tutto-parti in cui la classe ***tutto*** governa il periodo di vita delle classi ***parti***, che esistono esclusivamente come componenti del tutto. Questa associazione è nota come ***has-A***, perché un oggetto (componente) è parte di un altro oggetto di tipo composito. La notazione UML per le composizioni è un segmento che collega le due classi in cui l'estremo rivolto alla classe componente (*parte*) è semplice, mentre l'estremo rivolto alla classe composita (*tutto*) è caratterizzato da un piccolo rombo pieno.

Agli estremi del collegamento può essere specificata la molteplicità dell'associazione, ovvero il numero di istanze della classe associate a ciascuna delle istanze della classe che si trova all'estremo opposto.

ESEMPIO Il diagramma delle classi UML di FIGURA **13** rappresenta un'associazione di tipo compositivo dove viene stabilito che un oggetto di tipo *Automobile* **ha** un oggetto di tipo *Motore*.
La molteplicità 1 a 1 tra automobile e motore è data dal fatto che normalmente ogni automobile ha un motore e che ogni motore è di una sola automobile.

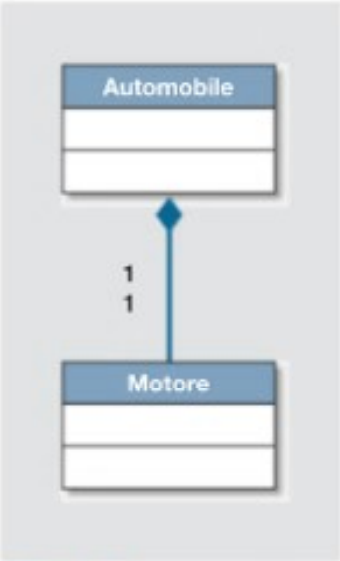


FIGURA **13**

Se il collegamento verso la classe componente è rappresentato da una freccia con l'estremità aperta, esso rappresenta un indicatore di «navigabilità» che indica la possibilità di accedere alla/e istanza/e della classe puntata dalla freccia a partire dall'istanza della classe che si trova all'altro estremo della freccia (questo normalmente significa che nella classe da cui parte la freccia ci sono uno o più attributi del tipo della classe a cui punta la freccia) (FIGURA **14**).

Non sempre è facile capire se utilizzare un'associazione di tipo «*is-A*» (generalizzazione) o di tipo «*has-A*» (composizione). Per esempio, dovendo progettare una classe *Radio*, a partire dalle classi esistenti *Sintonizzatore* e *DispositivoElettrico*, è abbastanza ovvio che *Radio* debba derivare da *DispositivoElettrico*; ma non è altrettanto ovvio se essa debba anche derivare da *Sintonizzatore*, oppure avere come attributo un'istanza di *Sintonizzatore* (FIGURA **15**).
Una prima decisione dipende dalla molteplicità dell'associazione tra le due classi: se non è 1 a 1 – cioè se una radio può avere più di un sintonizzatore – è sicuramente da escludere la derivazione; in caso contrario, la scelta in genere è determinata dal contesto d'uso della classe, e in particolare dal vantaggio/svantaggio derivante dal fatto che un oggetto di classe *Radio* sia anche – mediante il ricorso al polimorfismo – un oggetto di classe *Sintonizzatore*.

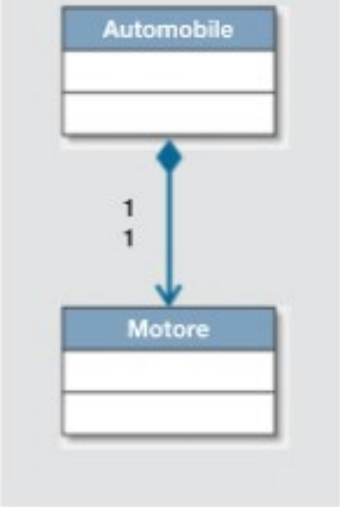


FIGURA **14**

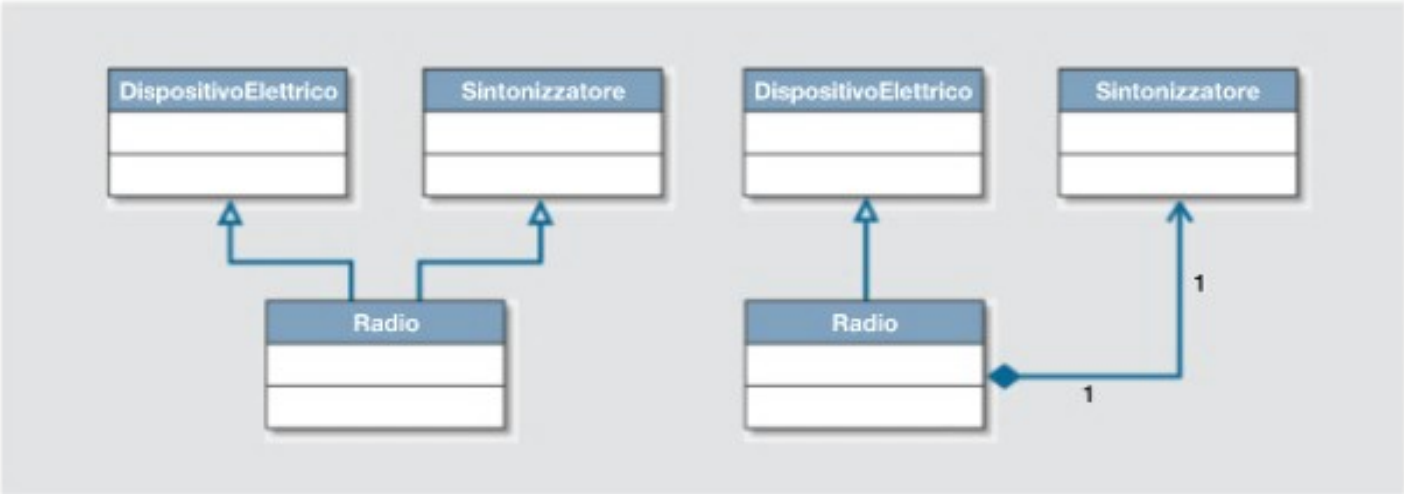


FIGURA **15**

5.4 Aggregazione

L'associazione di **aggregazione** è un'associazione tutto-parti, ma meno forte della composizione: una parte, infatti, può appartenere a più di una classe *tutto*, che può esistere anche indipendentemente dalle parti. La notazione UML per le aggregazioni è un segmento che collega le due classi in cui l'estremo rivolto alla classe aggregata (*parte*) è semplice, mentre l'estremo rivolto alla classe aggregante (*tutto*) è caratterizzato da un piccolo rombo vuoto.

ESEMPIO Il diagramma delle classi UML di FIGURA **16** rappresenta un'associazione di tipo aggregativo dove la classe *Parcheggio* aggrega la classe *Automobile*, perché un parcheggio (il tutto) aggrega più automobili (le parti), pur esistendo indipendentemente da esse.
La molteplicità indicata per l'associazione – un parcheggio (1) per zero o più automobili (0..*) – riflette il fatto che il parcheggio può essere vuoto, o contenere una o più automobili. Potrebbe essere esplicitato il segnalatore di navigabilità ponendo una freccia aperta sul lato del collegamento che punta alla classe *Automobile* per indicare che la classe *Parcheggio* ha come attributo una «collezione» di oggetti di tipo *Automobile*.

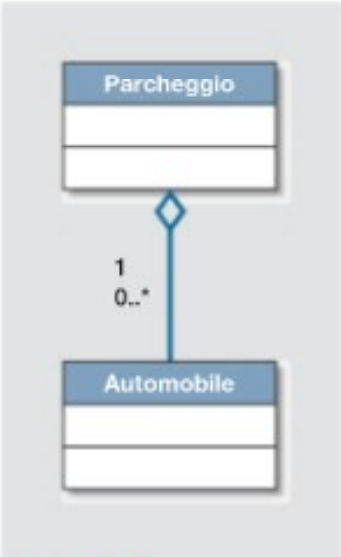


FIGURA **16**

Non sempre è semplice distinguere correttamente un'associazione di composizione da una di aggregazione: alcune situazioni possono infatti trarre in inganno. Un possibile criterio per distinguere l'aggregazione dalla composizione può essere quello di verificare se l'eliminazione di un oggetto composto implica naturalmente anche l'eliminazione dei suoi componenti: in caso affermativo si tratta di composizione, altrimenti di aggregazione.

I CONCETTI CHIAVE

OOP (*OBJECT ORIENTED PROGRAMMING*). La programmazione orientata agli oggetti è un paradigma di programmazione che prevede la definizione di oggetti software che interagiscono tra di loro attraverso lo scambio di messaggi.

ADT (*ABSTRACT DATA TYPE*). L'espressione *tipo di dato astratto* si riferisce a un tipo di dato completamente specificato, ma indipendente da una sua particolare implementazione; l'implementazione di un ADT è demandata allo sviluppatore del software sia per il contenuto informativo (*dati* o *attributi*), sia per le operazioni specifiche associate al tipo di dato (*metodi*). Un ADT è caratterizzato dalla definizione di un nuovo tipo di dato, dalla definizione di un insieme di operazioni sul tipo di dato il cui comportamento è definito mediante condizioni prestabilite e che costituiscono la sua interfaccia, essendo l'unico meccanismo tramite il quale è possibile interagire con il dato stesso.

INFORMATION HIDING. Il principio fondamentale di progettazione degli ADT è quello dell'*incapsulamento* che stabilisce la netta separazione tra l'interfaccia esposta da un ADT all'esterno per essere utilizzato e la sua implementazione interna che deve invece rimanere nascosta. Un obiettivo importante del principio di *information hiding* è rendere invisibili all'esterno di un ADT le scelte implementative che possono essere soggette a modifiche, proteggendo in questo modo il codice che utilizza l'ADT dalle conseguenze dovute all'eventuale cambiamento di tali scelte. L'incapsulamento, inoltre, garantisce un corretto utilizzo del tipo di dato prevenendo eventuali errori nella sua gestione da parte del codice che lo utilizza. Nella pratica della programmazione il principio di *information hiding* viene implementato ricorrendo a una tecnica che, in generale, prevede la classificazione degli attributi e dei metodi in *pubblici* o *privati*.

CLASSE. Nel contesto della OOD/OOP una classe rappresenta un modello formale per la descrizione di un certo tipo di oggetti definendone gli attributi, i metodi e le caratteristiche della sua interfaccia. A partire dalla definizione di una classe è possibile creare (*istanziare*) oggetti simili che condividono lo stesso insieme di attributi, lo stesso insieme di metodi e la stessa interfaccia.

OGGETTO. Istanza di una classe che rappresenta un'entità autonoma con propri valori per gli attributi.

COSTRUTTORE. Metodo speciale mediante il quale si istanziano oggetti della classe.

UML (*UNIFIED MODELING LANGUAGE*). Formalismo grafico per la progettazione e la documentazione del software che rende disponibili diversi tipi di diagrammi per descrivere i vari aspetti di un sistema OO (diagramma delle classi, diagramma degli oggetti, diagramma dei casi d'uso, diagramma di sequenza, ...).

DIAGRAMMA DELLE CLASSI. Diagramma UML che consente di rappresentare graficamente una o più classi con le loro componenti (attributi e metodi) e le relazioni che intercorrono tra le classi.

DIAGRAMMA DI SEQUENZA. Diagramma UML che descrive uno scenario di interazione tra oggetti mediante scambio di messaggi nel contesto di una determinata sequenza di azioni.

EREDITARIETÀ. Meccanismo dei linguaggi di programmazione OO che permette la creazione di nuove classi (*classi derivate* o *sottoclassi*) a partire da classi preesistenti (*classi base* o *superclassi*), ereditandone le caratteristiche (attributi e/o metodi), aggiungendone di nuove o ridefinendone altre. L'ereditarietà è finalizzata alla creazione di gerarchie di classi e grazie a essa si estende la possibilità di riutilizzare componenti comuni a più classi della stessa gerarchia. L'ereditarietà è *singola* se una sottoclasse deriva al più da una superclasse, *multipla* se una sottoclasse deriva da due o più superclassi.

POLIMORFISMO. È un concetto tipico della OOP e strettamente collegato all'ereditarietà tra classi; permette di ottenere comportamenti e risultati diversi invocando gli stessi metodi a carico di oggetti diversi. Il polimorfismo si basa sulla possibilità di ridefinire nelle classi derivate i comportamenti originali dei metodi ereditati dalla classe base per ottenere metodi in grado di operare in modo appropriato su oggetti di tipologia differente: sarà compito dell'ambiente di esecuzione decidere di volta in volta qual è il metodo corretto da utilizzare sulla base della specifica classe di cui l'oggetto è istanza. Mediante il polimorfismo è possibile trattare gli oggetti istanza delle classi di una gerarchia derivata da una classe base come se fossero istanze della classe base stessa.

CLASSI ASTRATTE. Una classe astratta si limita a definire l'interfaccia (la *firma*) di alcuni metodi dichiarandoli, ma non implementandoli. Queste operazioni saranno obbligatoriamente implementate dalle classi derivate. In questo modo una classe base astratta *fattorizza* le operazioni comuni a tutte le sue sottoclassi: una classe astratta non viene creata per istanziare oggetti, ma esclusivamente per derivarne altre classi che specificheranno le operazioni che essa dichiara. È infatti impossibile istanziare oggetti a partire da una classe astratta in quanto, essendovi metodi «lasciati in bianco» (metodi astratti), la loro eventuale invocazione non potrebbe essere eseguita.

ASSOCIAZIONI TRA CLASSI. Uno dei punti fondamentali della progettazione software OO consiste nel determinare come gli oggetti creati a partire dalle classi definite collaborino tra di loro. Nella programmazione OO gli oggetti interagiscono tra loro scambiando i messaggi che richiedono l'esecuzione di specifici metodi. Oltre al fatto che gli attributi di una classe possono essere del tipo definito da un'altra classe (*associazione statica*), sono gli scambi di messaggi che veicolano con i loro parametri le relazioni tra classi (*associazioni dinamiche*), la cui classificazione e documentazione è fondamentale nella fase di progettazione del software.

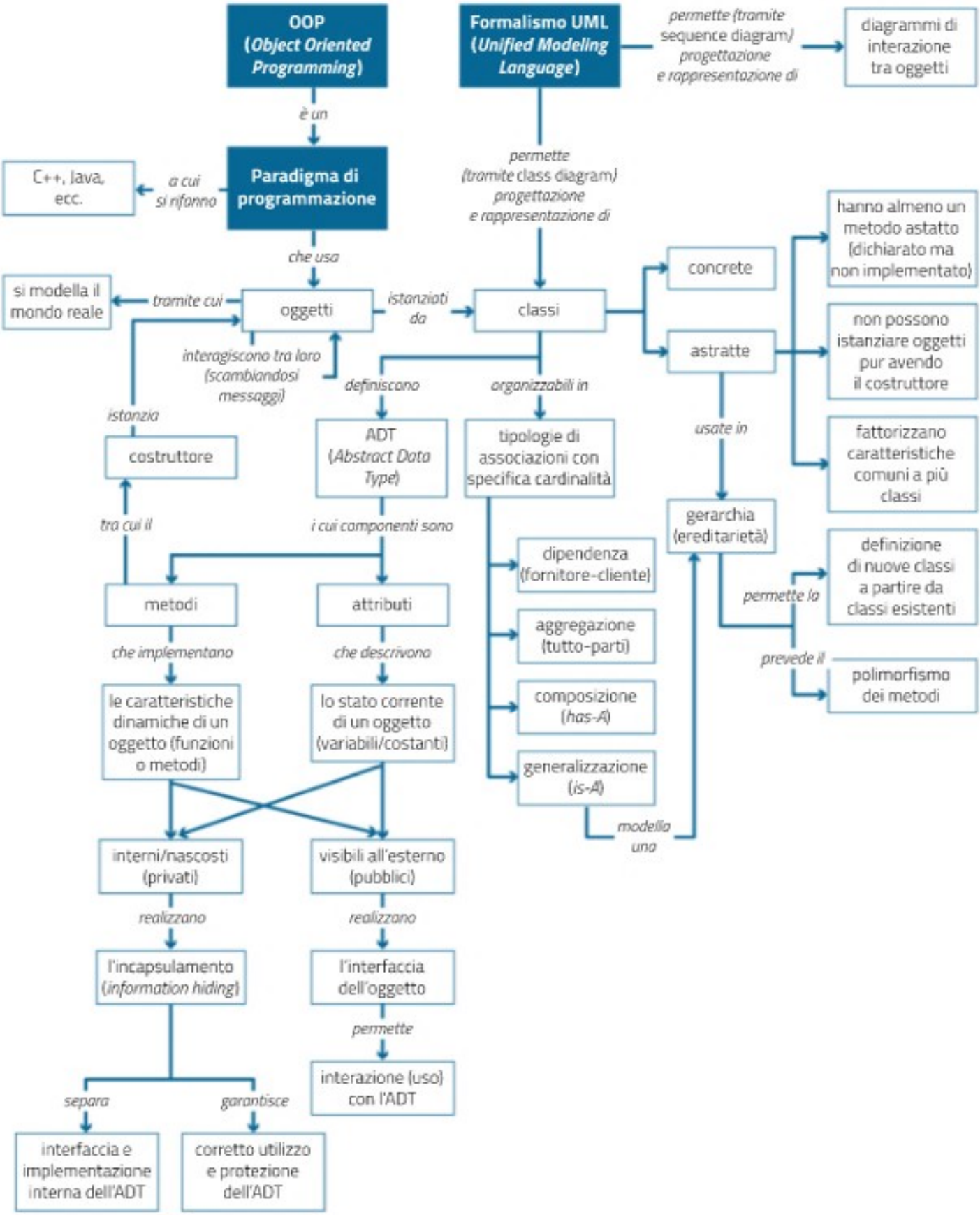
DIPENDENZA. La dipendenza, o *associazione d'uso*, è un tipo di associazione in cui le eventuali variazioni a un elemento (il *fornitore*) possono avere influenza sull'altro elemento (il *cliente*).

GENERALIZZAZIONE. L'ereditarietà realizzata dai linguaggi di programmazione OO è un'associazione tra classi molto comune ed è nota come *is-A*, perché un oggetto istanza di una sottoclasse è al tempo stesso istanza della relativa superclasse (polimorfismo).

COMPOSIZIONE. Associazione tra classi del tipo tutto-parti in cui la classe *tutto* governa il periodo di vita delle classi *parti*, che esistono esclusivamente come componenti del tutto. Questa associazione è nota come *has-A*, perché un oggetto (componente) è parte di un altro oggetto di tipo composito.

AGGREGAZIONE. Associazione tra classi del tipo tutto-parti, ma meno forte della composizione: una parte, infatti, può appartenere a più di una classe *tutto*, che può esistere anche indipendentemente dalle parti.

RIPASSA CON LA MAPPA



18.7.1 Association

An association denotes a relationship between classes and can be bidirectionally implied. Class attributes and multiplicities can be included at the target end(s).

18.7.2 Direct Association

Direct association, also known as navigability, is a relationship directing the source class to the target class. This relationship may be read Orchestra "has-a" Clarinet. Class attributes and multiplicities can be included at the target end. Navigability can be bidirectional between classes.

18.7.3 Composition Association

Composition association, also known as containment, models a whole-part relationship, where the whole governs the lifetime of the parts. The parts cannot exist except as components of the whole. This is a stronger form of association than aggregation. You could say a Score is "composed-of" one or more part(s).

18.7.4 Aggregation Association

Aggregation association models a whole-part relationship where the parts may exist independently of the whole. The whole does not govern the existence of the parts. You could say Orchestra is the whole and Clarinet is "part-of" Orchestra.

18.7.5 Temporary Association

Temporary association, better known as dependency, is represented where one class requires the existence of another class. It's also seen in cases where an object is used as a local variable, return value, or a member function argument. Passing a frequency to a tune method of class Clarinet can be read as class Clarinet depends on class Frequency, or Clarinet "uses-a" Frequency.

18.7.6 Generalisation

Generalisation is where a specialised class inherits elements of a more general class. In Java, we know this as inheritance, such as class extends class Woodwind, or Clarinet "is-a(n)" Woodwind.

18.7.7 Realisation

Realisation models a class implementing an interface, such as class Clarinet implements interface Instrument.

[Robert Liguori and Patricia Liguori, *Java Pocket Guide*, O'Reilly, 2008]

QUESTIONS

a **What is UML?**
b **How is a class diagram structured in UML?**
c **What does aggregation association mean?**
d **What does generalisation mean?**

A2

Il linguaggio di programmazione Java

Nel capitolo precedente abbiamo visto il diagramma UML della classe *Televisore*, una sua implementazione in linguaggio Java potrebbe essere la seguente:

```
public class Televisore {
  // attributi caratteristiche apparecchio
  private int pollici;
  private String schermo;
  private String colore;
  // attributi stato apparecchio
  private int canale;
  private int volume;
  private int luminosita;
  private boolean acceso;
  // costruttore
  public Televisore(int pollici, String schermo, String colore) {
    setPollici(pollici);
    setSchermo(schermo);
    setColore(colore);
    canale = 1;
    volume = 10;
    luminosita = 40;
    acceso = false;
  }
  // getter/setter
  public  int     getPollici() {return pollici;}
  private void    setPollici(int p) {pollici = p;}
  public  String  getColore() {return colore;}
  private void    setColore(String c) {colore = c;}
  public  String  getSchermo() {return schermo;}
  private void    setSchermo(String s) {schermo = s;}
  public  int     getCanale() {return canale;}
  public  int     getVolume() {return volume;}
  public  int     getLuminosita() {return luminosita;}
  public  boolean isAcceso() {return acceso;}
  public  void    setCanale(int c) {
    if (c>0 && c<99) canale = c;
  }
  // operazioni
  public  void    accendi() {acceso = true;}
  public  void    spegni() {acceso = false;}
  public  void    canaleSuccessivo() {
    if (canale<99) canale++;
  }
  public  void    canalePrecedente() {
```

```
    if (canale>0) canale--;
  }
  public  void    aumentaVolume() {
    if (volume<50) volume++;
  }
  public  void    diminuisciVolume() {
    if (volume>0) volume--;
  }
  public  void    aumentaLuminosita() {
    if (luminosita<80) luminosita++;
  }
  public  void    diminuisciLuminosita() {
    if (luminosita>0) luminosita--;
  }
}
```

Anche se non abbiamo ancora introdotto le caratteristiche di Java, confrontando questo codice con il diagramma UML della classe, è possibile riscontrare come gli attributi e le operazioni della classe *Televisore* siano state tradotte nella specifica sintassi del linguaggio.

- Sono stati definiti due tipi di attributi: quelli che descrivono le **caratteristiche** del televisore (colore esterno, tipologia e dimensione in pollici dello schermo) e quelli che descrivono lo **stato** del televisore in un determinato momento (acceso/spento, canale selezionato, volume impostato, livello di luminosità). Gli attributi sono stati definiti tutti di tipo **privato** per evitare che il codice esterno alla classe ne possa modificare il valore direttamente; per ognuno di essi sono stati previsti **metodi** che permettono la loro gestione controllata (per esempio l'impostazione del volume è limitata ai valori compresi tra 0 e 50, mentre il livello della luminosità tra 0 e 80).
- Esiste un metodo particolare che ha lo stesso nome della classe (*Televisore*): questo è il **costruttore** della classe, ovvero il metodo che viene invocato automaticamente quando si instanzia un oggetto a partire dalla classe stessa. Il costruttore ha in questo caso tre parametri, relativi rispettivamente a ognuna delle caratteristiche di uno specifico televisore: all'atto della creazione di un oggetto di classe *Televisore* sarà necessario specificare il numero di pollici, la tipologia di schermo (LCD, LED, ...), e il colore esterno (Nero, Bianco, ...).

  OSSERVAZIONE I valori degli attributi che definiscono lo **stato** sono inizializzati dal costruttore dell'oggetto a dei valori predefiniti: 1 per il canale, 10 per il volume, 40 per la luminosità e `false` per acceso.

- Alcuni metodi sono definiti di tipo `private`: essi non possono essere invocati direttamente dal codice esterno alla classe, ma sono richiamati solo da altri metodi della classe *Televisore* e possono essere più tolleranti sul controllo dei propri parametri (per esempio *setPollici*, che imposta la dimensione in pollici della diagonale dello schermo dell'oggetto *televisore*, viene invocato esclusivamente dal costruttore dal momento che, una volta creato un oggetto *televisore*, non avrebbe senso modificare questa caratteristica che non è soggetta a variare nel tempo!).

- Il nome di alcuni metodi inizia con il prefisso *get* o *set*: essi consentono rispettivamente di acquisire o impostare i valori degli specifici attributi degli oggetti di classe *Televisore*.

1 Caratteristiche, storia e applicazioni del linguaggio Java

Java è un linguaggio di programmazione orientato agli oggetti sviluppato sotto la guida di James Gosling in Sun Microsystems (in seguito acquisita da Oracle Corporation) a partire dalle ricerche effettuate presso l'università di Stanford in California agli inizi degli anni Novanta del secolo scorso. La sintassi di base del linguaggio (strutture di controllo, espressioni, ...) è volutamente simile a quella del linguaggio C/C++ allo scopo di favorire il passaggio a Java dei programmatori che lo utilizzavano in precedenza.

Concepito inizialmente come un linguaggio finalizzato allo sviluppo di applicazioni complesse per dispositivi elettronici *embedded*, ha avuto una grande diffusione a partire dal 1995 grazie all'espansione su scala mondiale della rete Internet come strumento di programmazione per l'ambiente web. La figura che segue raffigura il logo ufficiale del linguaggio di programmazione Java:

I punti di forza che hanno decretato il grande successo di questo linguaggio – ormai uno dei più utilizzati linguaggi di programmazione – sono i seguenti.

- **Orientamento agli oggetti.** Java è un linguaggio *object-oriented* e in quanto tale permette di sviluppare software che modella la realtà mediante entità (oggetti) dotate di specifiche d'uso e di funzionamento definite (classi); in ogni caso Java non è un linguaggio *object-oriented* «puro», per esempio i valori dei tipi primitivi, come i numeri interi, non sono oggetti.
- **Portabilità.** Il motto dei progettisti di Java è stato «*write once, run anywhere*» (letteralmente «scrivi una volta, esegui ovunque»): l'idea alla base di questo approccio è legata alla tecnologia denominata **JVM** (*Java Virtual Machine*) o **JRE** (*Java Runtime Environment*). Il compilatore Java, infatti, non produce codice eseguibile per una specifica piattaforma hardware/software, ma il cosiddetto *byte-code*, che viene interpretato dalla JVM. Ogni piattaforma hardware/software ha una propria JVM specifica, per cui il *byte-code* può essere indifferentemente eseguito su ciascuna piattaforma perché per ognuna di esse è la macchina virtuale a farsi carico della sua esecuzione.

Curiosità sul nome del linguaggio Java

Alcune voci mai confermate vogliono che il nome del linguaggio sia stato inteso dagli stessi creatori come acronimo per «*Just Another Vacuum Acronym*» («soltanto un altro acronimo vuoto») con ironico riferimento al grande numero di abbreviazioni utilizzate dagli sviluppatori software.
Dato che i creatori del linguaggio si riunivano spesso in un caffè dove discutevano del progetto, pare più probabile che il linguaggio abbia preso il nome dalla qualità di caffè Java dell'omonima isola dell'Indonesia.
A conferma di questa interpretazione vi è il logo ufficiale del linguaggio e il fatto che il *magic number* che identifica i file di *byte-code* (il formato eseguibile dei programmi Java) è in esadecimale CAFEBABE, cioè letteralmente «ragazza del caffè», forse un omaggio alla cameriera del caffè.

OSSERVAZIONE Queste caratteristiche hanno reso Java, fin dall'inizio, il linguaggio ideale per le applicazioni web: in questo contesto, infatti, non è dato conoscere allo sviluppatore la piattaforma hardware/software di esecuzione di un'applicazione che può di volta in volta essere diversa.

La FIGURA 1 esemplifica come un file di codice sorgente avente estensione «.java» viene trasformato dal compilatore in un file di *byte-code* con estensione «.class» che può essere eseguito su qualsiasi piattaforma hardware/software se dotata della specifica JVM.

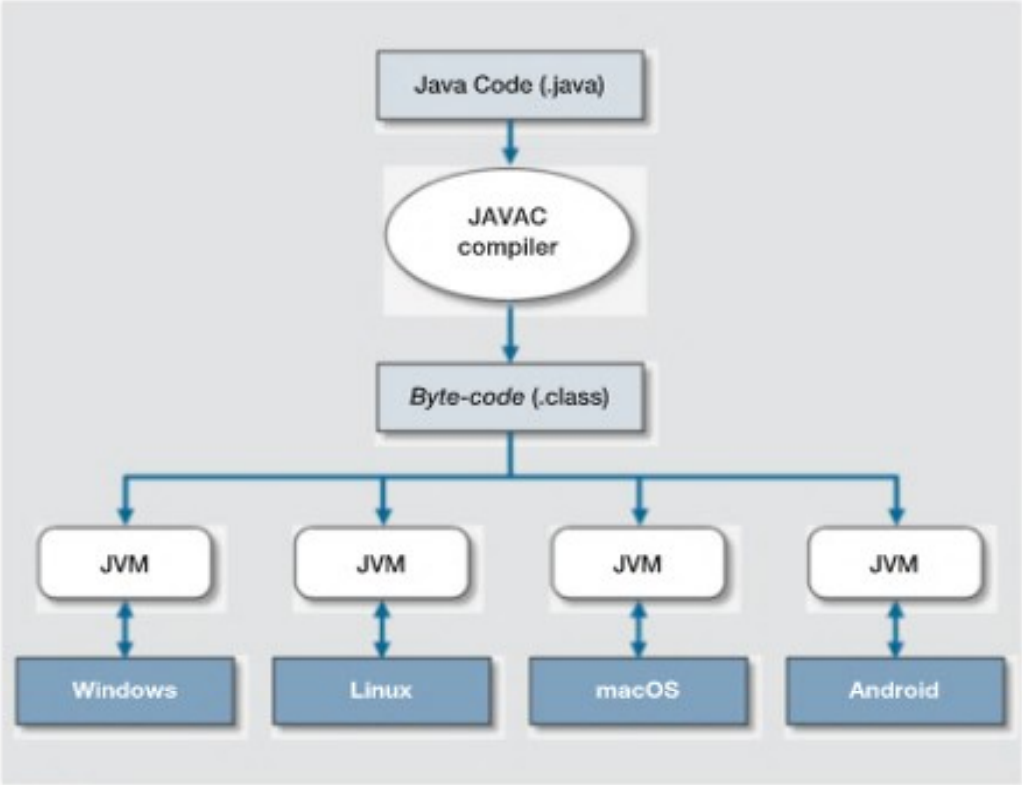


FIGURA 1

Byte-code

Il *byte-code* è un linguaggio intermedio – più astratto del linguaggio macchina di un processore reale – usato per definire le operazioni che costituiscono un programma.
Un linguaggio intermedio come il *byte-code* è utile nella implementazione dei linguaggi di programmazione perché riduce la dipendenza dall'hardware e facilita la creazione degli interpreti del linguaggio stesso.
Un programma in *byte-code*, infatti, viene «eseguito» da un altro programma che ne interpreta le istruzioni. Questo interprete è spesso indicato con il termine **macchina virtuale**, in quanto può essere visto come un computer astratto che simula buona parte delle funzionalità di un computer reale.
Questa astrazione offre la possibilità di scrivere programmi portabili in modo che risultino eseguibili in diversi ambienti operativi e con diverse architetture hardware.
Questo è un vantaggio che hanno anche i linguaggi di programmazione interpretati, tuttavia un interprete di *byte-code* risulta essere molto più performante di un interprete per un linguaggio di alto livello, perché ha poche e semplici istruzioni che simulano il funzionamento dell'hardware del computer.

In pratica la piena portabilità è un obiettivo tecnicamente difficile da raggiungere e il fatto che la tecnologia di Java sia riuscita o meno a centrarlo è un aspetto controverso. Sebbene sia possibile scrivere con il linguaggio Java programmi che si comportano in modo consistente quando sono eseguiti su piattaforme diverse, si deve tenere in considerazione che le stesse JVM sono programmi con i loro inevitabili *bug*, diversi per ciascun ambiente di esecuzione. Per questo motivo circola tra gli sviluppatori Java una parodia dello slogan originale che recita «*write once, debug anywhere*» (letteralmente «scrivi una volta, effettua il *debug* ovunque»).

- **Disponibilità di strumenti e librerie per la programmazione**. Uno degli aspetti che spesso portano all'adozione di Java come linguaggio per la realizzazione di un prodotto software è dato dalla quantità e qualità delle librerie di cui il linguaggio è dotato che lo rendono facilmente integrabile con altre tecnologie software (accesso a database relazionali, gestione di documenti XML, supporto nativo per la programmazione di rete e web sia lato client sia server, strumenti per la grafica e multimedialità, ...).

- **Sicurezza dell'esecuzione del codice.** L'ambiente di esecuzione della piattaforma Java è stato uno dei primi a consentire l'esecuzione automatica di codice proveniente da sorgenti remote: un codice potenzialmente incontrollato deve poter essere eseguito con la certezza che non possa interagire – per errore, o per intento malevolo – con l'ambiente del computer su cui viene eseguito.
- ***Multi-threading* nativo.** Il linguaggio di programmazione Java è stato uno dei primi a definire un modello di *multi-threading* nativo e indipendente dal sistema operativo di compilazione e di esecuzione; questa caratteristica originale, per il fatto che consente al programmatore di sfruttare i moderni processori *multi-core*, si sta dimostrando ogni giorno più importante.

2 Compilazione ed esecuzione di programmi Java; memoria *heap* e *garbage collector*

L'elemento fondamentale di un programma Java è la **classe**, un «modello» a partire dal quale è possibile istanziare oggetti indipendenti con caratteristiche simili. In una classe sono normalmente presenti due sezioni, una riservata alla componente informativa che si realizza nella definizione degli attributi, l'altra relativa all'aspetto operativo concretizzato nei metodi, ciascuno dei quali definisce una funzione. Dato che ogni oggetto è istanza di una classe, un programma Java è di fatto una collezione di oggetti che interagiscono mediante la loro interfaccia pubblica.

Il seguente programma Java visualizza la stringa *Hello, world!* ed è uno dei programmi più semplici che si possano realizzare:

```
public class Hello {
  public static void main (String[] args) {
    System.out.println("Hello, world!");
  }
}
```

Analizziamolo nel dettaglio:

- il nome della classe deve coincidere con quello del file in cui viene definita; la convenzione adottata da tutti i programmatori Java, e assunta dalla maggior parte degli ambienti di sviluppo, consiste nell'avere una sola classe per file il cui nome ha estensione «.java» (in questo caso *Hello.java*);
- un programma Java è costituito da un insieme di classi: nel nostro esempio abbiamo una sola classe la cui dichiarazione inizia con la parola riservata `class` seguita dal nome della classe – *Hello* – e da un simbolo «{» a cui corrisponde il simbolo «}» alla fine della dichiarazione della classe;
- la classe *Hello* non ha attributi e contiene un solo metodo, denominato ***main***; diversamente da quanto avviene nel linguaggio C++, in Java un

metodo può essere invocato solo da parte del codice di un altro metodo – della stessa classe, o di una classe distinta – per cui è necessario un metodo iniziale da cui ha inizio l'esecuzione del programma: il metodo iniziale è il metodo *main* che ha sempre la seguente firma:

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

- il metodo *main* si caratterizza per le seguenti proprietà:
  - è pubblico (`public`), ovvero visibile da ogni punto del codice;
  - è statico (`static`), cioè è invocabile indipendentemente dall'esistenza di oggetti istanza della classe, come in questo caso;
  - non restituisce nulla (`void`);
  - gli eventuali parametri in input forniti dall'utente sulla riga di comando costituiscono un vettore di stringhe (cioè di oggetti della classe predefinita *String*).

Ogni classe di un programma Java può avere un suo metodo *main*, in ogni caso l'esecuzione del programma ha inizio con l'invocazione del metodo *main* di una classe specificata come «classe principale» (*main class*).

Dopo aver editato il codice sorgente Java in un file di testo la cui denominazione rispetta la convenzione indicata è possibile compilare il codice eseguibile (*byte-code*) invocando il comando `javac` sulla riga di comando:

```
>javac Hello.java
```

Se il codice è sintatticamente corretto, il compilatore genera il file *Hello.class* che contiene il *byte-code*.

Per eseguire il programma è necessario eseguire la macchina virtuale (il programma *java*) e fornire come argomento il *byte-code* da interpretare:

```
>java Hello
```

Se non vi sono errori in fase di esecuzione viene visualizzato l'output del programma:

```
Hello, world!
```

La visualizzazione della stringa *Hello, world!* è causata dalla seguente riga di codice del metodo *main*:

```
System.out.println("Hello, world!");
```

che invoca il metodo *println* della classe predefinita *PrintStream* di cui è istanza l'attributo statico *out* della classe *System*; l'invocazione del metodo *print* al posto di *println* causa la visualizzazione della stringa fornita come argomento senza il ritorno a capo.

JRE e JDK

Per eseguire programmi Java su un computer è sufficiente installare il JRE che comprende la JVM per la specifica piattaforma hardware/software.
Per compilare programmi Java è indispensabile anche il JDK (*Java Development Kit*) che comprende il compilatore.
Entrambi sono liberamente e gratuitamente scaricabili dal sito dedicato della Oracle Corporation.
I programmatori Java utilizzano IDE (*Integrated Development Environment*) avanzati – i più diffusi sono *Eclipse* e *Netbeans* – ma anche in questo caso in fase di compilazione del codice viene richiamato il compilatore del JDK.

La JVM – l'ambiente di esecuzione dei programmi Java – è logicamente costituita dai seguenti componenti logici:

- un insieme di istruzioni (*byte-code*);
- un gruppo di *registri*;
- un'area di memoria per l'esecuzione dei metodi (*stack*);
- un'area di memoria per l'allocazione degli oggetti (*heap*) su cui opera lo strumento automatico di distruzione degli oggetti inutilizzati (*garbage collector*);
- un'*area* per la memorizzazione *dei metodi*.

Byte-code: l'insieme di istruzioni della macchina virtuale è ottimizzato in modo che risulti piccolo e compatto.

Registri: i registri della macchina virtuale sono analoghi ai registri che si trovano nel processore di un computer reale; essi contengono lo stato in cui si trova la macchina durante l'esecuzione del *byte-code*.

Stack: il funzionamento della macchina virtuale è basato sullo *stack* che viene utilizzato per passare i parametri ai metodi e per memorizzare temporaneamente i risultati restituiti dallo loro invocazione.

Heap: il termine si riferisce all'area di memoria nella quale sono allocati gli oggetti istanziati a partire dalle classi utilizzando la parola chiave `new` del linguaggio; all'inizio dell'esecuzione di un programma lo *heap* viene allocato in base a una dimensione predeterminata, ma nel corso dell'esecuzione la dimensione si adatta dinamicamente al numero e alla dimensione degli oggetti creati.

Una delle caratteristiche fondamentali del linguaggio Java è la rimozione automatica degli oggetti non più utilizzati liberando lo sviluppatore software da questa incombenza. Tutti gli oggetti istanziati in un programma Java sono individuati da un riferimento alla posizione che occupano nello *heap*: la JVM attiva periodicamente il ***garbage collector*** (letteralmente «raccoglitore di spazzatura») che è in grado di rilevare gli oggetti non più riferiti per recuperare l'area di memoria da essi occupata per renderla nuovamente disponibile.

Area dei metodi: l'area dei metodi contiene il *byte-code* dei metodi delle classi del programma Java in esecuzione.

Istanziando più oggetti a partire dalla stessa classe, l'occupazione della memoria cresce solo in relazione alla dimensione degli attributi **non statici** che sono replicati per ogni singolo oggetto; i metodi e gli attributi statici infatti sono memorizzati una sola volta essendo comuni a tutti gli oggetti istanza di una classe.

Se aggiungiamo il seguente metodo *main* alla classe *Televisore* introdotta all'inizio del capitolo possiamo istanziare un oggetto della classe per verificare la funzionalità di alcuni dei metodi:

```
public static void main (String[] args) {
  Televisore t;

  t = new Televisore(32,"LED","nero");
  t.canaleSuccessivo();
  t.aumentaVolume();
  System.out.println(t.getColore());
  System.out.println(t.getCanale());
  System.out.println(t.getVolume());
}
```

La dichiarazione `Televisore t;` consente di definire un riferimento a un oggetto istanza della classe *Televisore*, ma non istanzia un oggetto.

L'istruzione `t = new Televisore(32,"LED","nero");` istanzia un oggetto di classe *Televisore* nello *heap* che sarà riferito da *t* come illustrato in FIGURA 2.

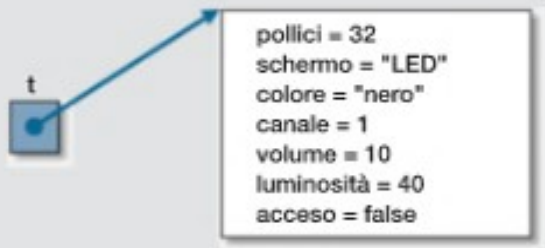


FIGURA 2

In Java un oggetto viene sempre creato mediante l'uso esplicito della parola chiave **new**; nell'esempio precedente la dichiarazione della variabile di tipo *Televisore* e la creazione di un oggetto di classe *Televisore* potevano anche essere formulate in un'unica riga di codice:

```
Televisore t = new Televisore(32,"LED","nero");
```

In Java non esiste il concetto esplicito di «puntatore», ma di fatto tutte le variabili cui è possibile assegnare oggetti istanza di classi sono puntatori impliciti.

Se nel metodo *main* della classe *Televisore* vi fossero le seguenti righe di codice

```
...
Televisore t1, t2;
t1 = new Televisore(32,"LED","nero");
t2 = new Televisore(27,"LCD","grigio");
...
```

la situazione nello *heap* sarebbe quella di FIGURA 3.

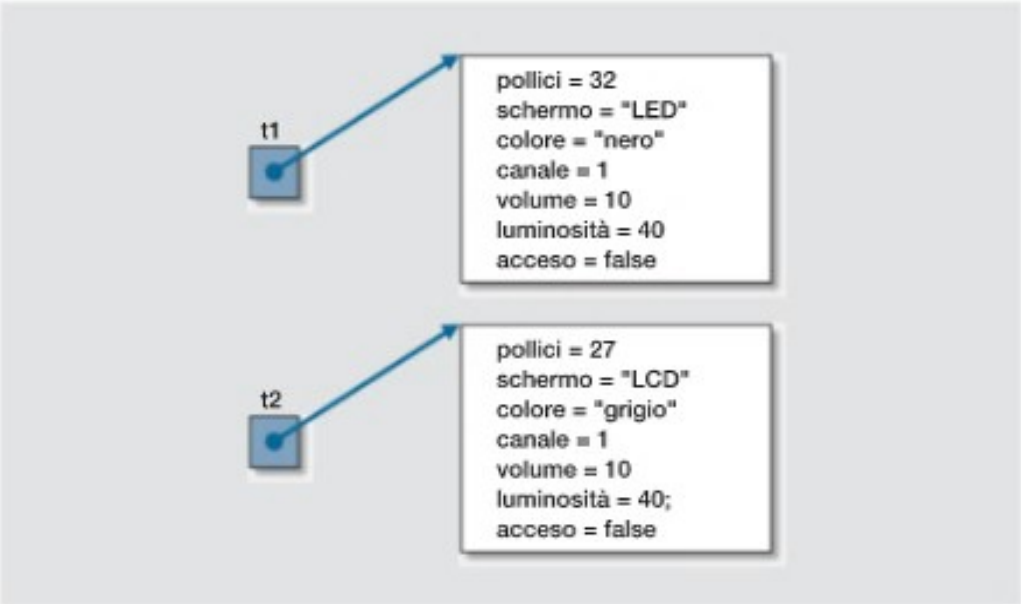


FIGURA 3

Ma nel caso in cui il codice fosse invece quello che segue:

```
...
Televisore t1, t2;
t1 = new Televisore(32,"LED","nero");
t2 = t1;
...
```

si avrebbe che le variabili *t1* e *t2* riferiscono lo stesso oggetto (l'unico che viene effettivamente creato) (FIGURA 4).

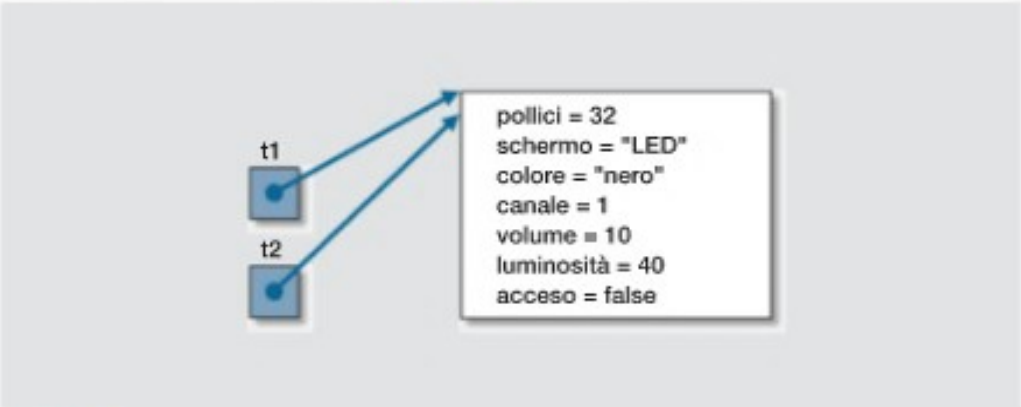


FIGURA 4

Tutte le variabili aventi come tipo una classe, come per esempio *t1* e *t2*, al momento della loro dichiarazione sono inizializzate con il valore `null`, che indica che non riferiscono alcun oggetto nello *heap*. Nell'esempio precedente la variabile *t1* assume come valore il riferimento all'oggetto creato nello *heap* solo al momento dell'esecuzione dell'istruzione

```
t1 = new Televisore(32,"LED","nero");
```

mentre *t2* assume il valore del riferimento allo stesso oggetto con l'esecuzione dell'istruzione:

```
t2 = t1;
```

Quando due variabili riferiscono lo stesso oggetto nello *heap*, la variazione degli attributi dell'oggetto è ovviamente visibile attraverso entrambi i riferimenti; per esempio il seguente frammento di codice nel metodo *main* degli esempi precedenti

```
...
Televisore t1, t2;
t1 = new Televisore(32,"LED","nero");
t2 = t1;
t1.aumentaVolume();
...
```

ha come conseguenza che anche il volume di *t2* vale 11 – e non più 10 come inizialmente impostato – in quanto è un attributo dello stesso oggetto che è stato modificato!

Le ultime istruzioni del metodo *main* della classe *Televisore*

```
...
t1.canaleSuccessivo();
t1.aumentaVolume();
System.out.println(t1.getColore());
System.out.println(t1.getCanale());
System.out.println(t1.getVolume());
...
```

producono a video il seguente output

```
nero
2
11
```

in quanto i valori degli attributi *canale* e *volume* sono incrementati dall'invocazione dei metodi *canaleSuccessivo* e *aumentaVolume*.

OSSERVAZIONE Si noti che il metodo *println*, pur accettando formalmente come argomento una stringa di caratteri, è in grado di visualizzare anche il contenuto di valori numerici. La giustificazione di questo comportamento potrà essere completamente chiarita solo dopo lo studio del polimorfismo nei linguaggi di programmazione orientati agli oggetti.

3 Struttura di un programma Java e fondamenti del linguaggio

Sotto l'aspetto sintattico un programma Java è un insieme di classi ognuna delle quali è definita in un file con estensione «.java» avente lo stesso nome della classe. Almeno la classe principale deve avere un metodo *main* da cui avrà inizio l'esecuzione del programma.

3.1 *Package*

Le classi che costituiscono un programma Java sono normalmente suddivise in gruppi denominati ***package***; il ricorso ai *package* permette di organizzare le classi del programma in modo ordinato: gli sviluppatori spesso utilizzano i *package* per raggruppare classi logicamente interdipendenti. Nei file delle classi che appartengono allo stesso *package* è riportata prima della loro definizione la seguente riga di codice

```
package nome_package;
```

dove *nome_package* è comune a tutte le classi del *package* e deve essere univoco almeno nel contesto del progetto software.

ESEMPIO Molti sviluppatori e aziende di produzione di software assicurano l'univocità dei nomi dei *package* che sviluppano utilizzando una gerarchia di questo tipo:

```
package us.nasa.software.spacecraft.game;
```

I *package* di classi, una volta compilati, possono essere memorizzati in file compressi con estensione «.JAR» che comprendono tutti i file di *byte-code* delle singole classi del *package*.

OSSERVAZIONE Un *package* definisce un unico spazio dei nomi per le classi che contiene: in questo modo sviluppatori diversi possono collaborare allo stesso progetto senza curarsi di eventuali sovrapposizioni dei nomi delle classi se i nomi di *package* che realizzano sono diversi.

3.2 Classi

Ogni file che contiene la definizione di una classe dovrebbe avere la seguente struttura di massima:

- commento di intestazione del file;
- dichiarazione del *package*;
- sezione di *import* dei *package*;
- definizione della classe:
  - attributi;
  - costruttori;
  - metodi.

L'importazione dei *package* necessari per la corretta compilazione della classe si effettua con la parola chiave `import`.

ESEMPIO Dovendo utilizzare la classi della libreria per l'uso dei file di testo, è necessario premettere alla definizione della classe le seguenti righe di codice:

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

È possibile dichiarare l'importazione di un intero gruppo di *package* utilizzando il simbolo «*»:

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

3.3 Membri di una classe

Gli *attributi*, sia variabili sia costanti, costituiscono il contenuto informativo di una classe: essi sono gli elementi che descriveranno lo stato degli oggetti istanziati a partire dalla classe.

OSSERVAZIONE Oltre agli attributi – o variabili di istanza – definiti a livello di classe, il linguaggio Java consente la definizione di variabili locali all'interno dei singoli metodi.

Le *costanti* sono identificate dalle parole chiave `static final`[1].

1. Usando solo `final` senza `static`, si definisce una variabile che potrà essere inizializzata una sola volta sia nella fase di dichiarazione o con una successiva assegnazione.

ESEMPIO Il valore di π può essere così definito:

```
static final double PI_GRECO = 3.1416;
```

I *metodi* realizzano i comportamenti che gli oggetti implementano: essi sono definiti in modo analogo alle funzioni del linguaggio C/C++, ma non si possono avere parametri passati per riferimento. Di fatto, nel caso di parametri istanze di classi predefinite o definite dal programmatore, si ha sempre a che fare con un riferimento implicito; l'eventuale variazione del parametro nel codice del metodo lo modifica anche nel contesto di invocazione del metodo.

Le *strutture di controllo del flusso di esecuzione* sono le stesse del linguaggio C/C++:

- `if-else`
- `switch-case`
- `while`
- `do-while`
- `for`

I membri di una classe – siano essi attributi o metodi – sono caratterizzati, oltre che dal loro tipo, anche dalla:

- visibilità data dal contesto di dichiarazione dell'attributo o del metodo o da un'eventuale omonimia di un parametro o di una variabile locale che lo nasconde;
- accessibilità stabilita tramite i modificatori `private`, `protected`, `public` e `package` (se non specificato si assume `package`).

La TABELLA 1 riporta quali metodi possono accedere agli attributi o invocare i metodi di una classe in base al tipo di accessibilità specificata nella dichiarazione.

TABELLA 1

Accessibilità	Metodi della classe	Metodi di una classe derivata	Metodi di una classe dello stesso *package*	Metodi di tutte le classi
`private`	SÌ	NO	NO	NO
`protected`	SÌ	SÌ	SÌ	NO
`public`	SÌ	SÌ	SÌ	SÌ
`default/package`	SÌ	NO	SÌ	NO

OSSERVAZIONE Anche se è possibile fare direttamente riferimento a un attributo pubblico di una classe direttamente nella forma *oggetto.attributo*, è generalmente preferibile definire tutti gli attributi come privati o protetti, lasciando che la loro gestione sia consentita esclusivamente in maniera controllata mediante specifici metodi.

3.4 Membri statici

Con il modificatore `static` è possibile qualificare attributi o metodi come appartenenti alla classe in cui sono definiti, invece che agli oggetti istanziati a partire dalla classe.

In particolare, se la dichiarazione di un attributo è preceduta dalla parola chiave `static`, esso è un attributo di classe e non di una determinata istanza di essa ed è condiviso da tutte le istanze della classe. Un'eventuale modifica al suo valore è quindi automaticamente condivisa da tutti gli oggetti istanziati a partire dalla classe.

ESEMPIO Il diagramma UML di FIGURA 5 rappresenta una classe *Libro* per la quale il prezzo di ogni libro è dato da un costo fisso costante di 5.5 € più un costo variabile in funzione delle pagine che lo compongono.

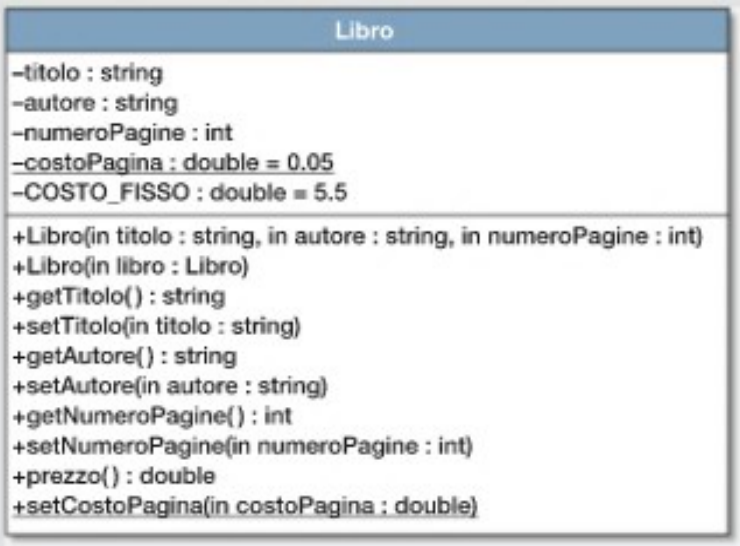


FIGURA 5

Gli attributi e i metodi sottolineati sono, in base alla convenzione UML, statici. La classe è implementata in linguaggio Java con il seguente codice:

```
public class Libro {
   private String titolo;
   private String autore;
   private int numeroPagine;
   private static double costoPagina = 0.05;
   final double COSTO_FISSO = 5.5;

public Libro(String titolo, String autore, int numeroPagine) {
   this.titolo = titolo;
   this.autore = autore;
   this.numeroPagine = numeroPagine;
}
public Libro(Libro libro) {
   this.titolo = libro.titolo;
   this.autore = libro.autore;
   this.numeroPagine = libro.numeroPagine;
}
public String getTitolo() {
   return titolo;
}
public void setTitolo(String titolo) {
   this.titolo = titolo;
}
public String getAutore() {
   return autore;
}
public void setAutore(String autore) {
   this.autore = autore;
}
public int getNumeroPagine() {
   return numeroPagine;
}
public void setNumeroPagine(int numeroPagine) {
   this.numeroPagine = numeroPagine;
}
public double prezzo() {
   return COSTO_FISSO + numeroPagine*costoPagina;
}
public static void setCostoPagina(double costo){
   costoPagina = costo;
}
}
```

Se si istanziano due oggetti distinti con il seguente frammento di codice

```
Libro l1 = new Libro("Pinocchio", "C. Collodi", 150);
Libro l2 = new Libro("Pollicino", "C. Perrault", 80);
```

le istruzioni

```
System.out.println(l1.getTitolo()+": "+l1.prezzo());
System.out.println(l2.getTitolo()+": "+l2.prezzo());
```

producono il seguente output:

```
Pinocchio: 13.0
Pollicino: 9.5
```

Se ripetiamo le stesse istruzioni dopo aver modificato il costo della pagina

```
Libro.setCostoPagina(0.1);
System.out.println(l1.getTitolo()+": "+l1.prezzo());
System.out.println(l2.getTitolo()+": "+l2.prezzo());
```

si ottiene in output la corrispondente variazione per entrambi i libri

```
Pinocchio: 20.0
Pollicino: 13.5
```

perché il metodo statico *setCostoPagina* ha modificato l'attributo statico *costoPagina* condiviso da tutte le istanze della classe *Libro*.
Nel codice precedente il metodo statico è stato invocato a partire dall'identificatore della classe stessa, ma è possibile farvi riferimento da un qualsiasi oggetto istanza della classe ottenendo esattamente lo stesso risultato:

```
l1.setCostoPagina(0.1);
```

Nell'esempio illustrato è stata utilizzata la parola chiave `this`, il cui significato è quello di fare esplicito riferimento all'oggetto corrente.

Nel caso specifico ha permesso di riferire gli attributi dei parametri del costruttore per i quali sono stati usati gli stessi nomi; l'uso del riferimento `this` ha evitato l'ambiguità tra il parametro e l'attributo, pertanto l'assegnazione

```
this.titolo = titolo;
```

viene interpretata come assegnamento del valore del parametro *titolo* all'attributo *titolo* dell'oggetto corrente identificato da `this.titolo`.

Analogamente agli attributi, un metodo statico è un metodo della classe che può essere invocato indipendentemente dagli oggetti istanziati. In generale, per invocare un metodo si riferisce un'istanza della classe con la notazione *oggetto.metodo*. Questa notazione può rappresentare una limitazione perché alcuni metodi sono semplici funzioni che restituiscono un output a partire dall'input fornito senza interessare lo stato dell'oggetto. È possibile dichiarare metodi statici che non richiedono la creazione di un oggetto per essere invocati: metodi di questo tipo possono infatti essere invocati utilizzando direttamente il nome della classe, senza che occorra applicarli ad alcuna istanza e in pratica corrispondono alle funzioni del linguaggio C/C++.

ESEMPIO L'istruzione

```
int n = Integer.parseInt("123");
```

assegna alla variabile *n* il valore 123 restituito dal metodo statico *parseInt* della classe *Integer* che trasforma in un valore numerico il contenuto di una stringa di caratteri.

La strutturazione del linguaggio Java obbliga a definire i metodi statici all'interno di una classe, ma per invocarli è sufficiente premettere il nome della classe al loro nome.

I metodi statici hanno comunque un'importante ma giustificata limitazione: non possono accedere agli attributi non statici della classe in quanto questi assumono valori specifici per ciascuna istanza.

OSSERVAZIONE Il metodo *main* di una classe è statico: infatti l'interprete del linguaggio Java inizia l'esecuzione di un programma a partire dal nome della classe principale di cui inizialmente non viene creata alcuna istanza.

3.5 Costruttori

Tra i metodi di una classe ne esiste uno, o – come vedremo in seguito – più di uno, che ha lo stesso nome della classe: il **metodo costruttore** viene invocato automaticamente al momento della creazione di un oggetto istanza della classe utilizzando l'operatore `new`. La sua funzione principale è quella di assegnare un valore iniziale agli attributi del nuovo oggetto creato.

Anche se il linguaggio Java non richiede esplicitamente la sua dichiarazione (viene in questo caso invocato un costruttore implicito) è buona norma prevederlo per ogni classe, in modo da assicurare la corretta inizializzazione di un oggetto all'atto della sua creazione. Il costruttore non ha tipo di ritorno perché l'oggetto che restituisce assume sempre il tipo della classe stessa e il suo livello di accessibilità in generale è `public`. (Esistono casi in cui, volendo negare la creazione incontrollata di oggetti, il costruttore viene dichiarato `private`.)

Una classe può avere più di un costruttore se questi si differenziano nella firma (*overloading*): il compilatore selezionerà automaticamente il costruttore corretto in funzione del numero e del tipo dei parametri forniti nell'invocazione.

Una buona pratica di programmazione prevede che ogni oggetto abbia due costruttori particolari.

- Il **costruttore di copia** è un costruttore a cui viene fornito come argomento un oggetto istanza della stessa classe per realizzarne un clone, copiandone i singoli attributi.

ESEMPIO Il costruttore di copia della classe *Libro* di un esempio precedente può essere utilizzato per copiare un oggetto istanza della classe:

```
Libro l1 = new Libro("L'isola misteriosa", "J. Verne", 500);
Libro l2 = new Libro(l1);
```

Si noti che nello *heap* sono stati creati due oggetti distinti, copia uno dell'altro, ma indipendenti rispetto alle successive variazioni.

Javabeans

Una classe Java che rispetti le seguenti convenzioni:

- ha un costruttore pubblico senza parametri;
- gli attributi sono accessibili mediante metodi *get/set*;
- è serializzabile[1];

è un componente software riutilizzabile in contesti diversi comunemente definito *Javabean*.

1. Questo concetto sarà illustrato nel prossimo capitolo.

- Il **costruttore di default**, privo di argomenti, che inizializza gli attributi a un valore predefinito.

ESEMPIO

Il costruttore di default della classe *Libro* dell'esempio precedente può essere così implementato:

```
public Libro() {
   this.titolo = "";
   this.autore = "";
   this.numeroPagine = 0;
}
```

Java Enterprise e POJO

La diffusione del linguaggio di programmazione Java è in larga parte dovuta al suo impiego in applicazioni server di tipo *enterprise*, per le quali è indispensabile seguire rigorose convenzioni di codifica e di implementazione, che riguardano in particolare i cosiddetti componenti software EJB (*Enterprise JavaBean*).
In un mondo di acronimi Martin Flower ha un po' ironicamente coniato il termine POJO (*Plain Old Java Object*) per identificare le normali classi Java come quelle che presentiamo in questo corso.

4 La struttura di base di una classe

Come abbiamo detto, un programma Java è un insieme di classi ciascuna delle quali è memorizzata in un file con estensione «.java» avente lo stesso nome della classe. La classe principale del programma deve prevedere un metodo *main* da cui avrà inizio l'esecuzione. Una buona pratica di programmazione prevede comunque di dotare ogni singola classe sviluppata di un metodo *main* che esegua un test dei metodi della classe, indipendentemente dal contesto di utilizzazione finale; senza sostituire la necessaria documentazione della classe, il metodo *main* costituisce anche un esempio di invocazione dei metodi per gli sviluppatori che devono utilizzare la classe nei propri programmi.

Il diagramma UML di FIGURA 6 che segue è relativo alla classe *Punto*, che permette di istanziare oggetti che rappresentano ciascuno un punto nel piano cartesiano.

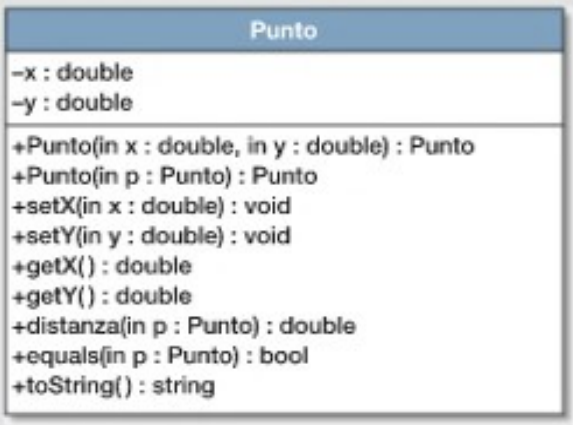


FIGURA 6

Il codice Java che implementa la classe è il seguente:

```
1  package geometria;
2
3  public class Punto {
4    private double x;
```

```
5     private double y;
6
7     public Punto(double x, double y) {
8       setX(x);
9       setY(y);
10    }
11
12    public Punto(Punto p) {
13      x = p.getX();
14      y = p.getY();
15    }
16
17  public void setX(double x) { this.x = x; }
18  public void setY(double y) { this.y = y; }
19  public double getX() { return x; }
20  public double getY() { return y; }
21
22  public double distanza(Punto p) {
23    double dx = x - p.getX();
24    double dy = y - p.getY();
25    return Math.sqrt((dx*dx)+(dy*dy));
26  }
27
28  public boolean equals(Punto p) {
29    return ((x==p.x) && (y==p.y));
30  }
31
32    public String toString() { return "("+x+";"+y+")"; }
33
34    public static void main(String[] args) {
35      Punto p1 = new Punto(1.,1.);
36      Punto p2 = new Punto(2.,2.);
37      Punto p3 = new Punto(p1);
38      System.out.println("P1="+p1.toString());
39      System.out.println("P2="+p2.toString());
40      System.out.println("P3="+p3.toString());
41      System.out.println("Distanza P1-P2: "+p1.distanza(p2));
42      System.out.println("Distanza P1-P3: "+p1.distanza(p3));
43      if (p1.equals(p3))
42        System.out.println("P1 e P3 coincidono");
43      else
44        System.out.println("P1 e P3 non coincidono");
45    }
46  }
```

che produce in output:

```
P1=(1.0;1.0)
P2=(2.0;2.0)
P3=(1.0;1.0)
Distanza P1-P2: 1.4142135623730951
Distanza P1-P3: 0.0
P1 e P3 coincidono
```

Analizziamo i dettagli del codice (tra parentesi sono indicati i numeri di riga a cui si fa riferimento).

- Gli attributi della classe sono due (4, 5): la coordinata *x* e la coordinata *y* del punto nel piano cartesiano, per entrambi esistono i metodi di accesso (*getter*) e modificatori (*setter*).
- Esistono due distinti costruttori (7-10 e 12-15): il primo istanzia un punto a partire dalle coordinate *x* e *y* fornite come parametri; il secondo istanzia un punto a partire dalle coordinate di un altro punto fornito come parametro.

  OSSERVAZIONE Il secondo costruttore è un costruttore di copia. Il compilatore Java seleziona il costruttore corretto in funzione dei parametri passati al momento dell'invocazione: il primo viene scelto se sono forniti due parametri di tipo `double`, mentre il secondo se è fornito come argomento un oggetto di tipo *Punto*.

Il costruttore di copia **istanzia un nuovo oggetto** punto i cui attributi hanno lo stesso valore di quelli dell'oggetto passato come parametro.

- Per determinare la distanza tra due punti si è fatto ricorso al metodo statico *sqrt* della classe di libreria *Math* per calcolare la radice quadrata (25).
- È stato definito il metodo *toString* che trasforma in una stringa di caratteri opportunamente formattata il valore degli attributi della classe (32).

Questo metodo sarebbe stato comunque disponibile per gli oggetti di classe *Punto*, come per qualsiasi altro oggetto.
Tutte le classi dichiarate in Java, infatti, ereditano dalla classe predefinita *Object*, che definisce un formato standard per alcuni metodi di utilità generale tra cui *toString*. Il comportamento del metodo originale deve però essere sempre ridefinito in base agli attributi specifici della classe.

- È stato definito il metodo *equals*, che verifica se un punto coincide con un diverso punto fornito come parametro (28-30).

  OSSERVAZIONE Anche il metodo di utilità *equals* viene ereditato dalla classe *Object*, ma è necessario ridefinirlo in base alle caratteristiche della classe, come nel caso specifico. Infatti, nel caso in cui esso non sia ridefinito, il suo esito coincide con quello dell'operatore di confronto «==», che risulta vero solo se due oggetti sono in realtà due riferimenti allo stesso oggetto nello *heap*: non è questo che il programmatore di solito intende per uguaglianza di due oggetti!

- È stato definito il metodo *main*, che consente di verificare la funzionalità dei metodi della classe indipendentemente da altre classi (34-45).

  OSSERVAZIONE La pratica di prevedere un metodo *main* per effettuare un test dei metodi della classe è valida solo per classi piuttosto semplici. Avendo a che fare con classi più complesse è preferibile definire una

classe distinta di test con un proprio metodo *main* che istanzi uno o più oggetti della classe da testare.

Proseguiamo l'esempio definendo la classe *Triangolo*, che rappresenta un triangolo nel piano cartesiano utilizzando oggetti di classe *Punto* per definirne i vertici. Il diagramma UML delle due classi esplicita un'associazione di tipo compositivo in quanto un triangolo **ha** tre punti come vertici (FIGURA **7**).

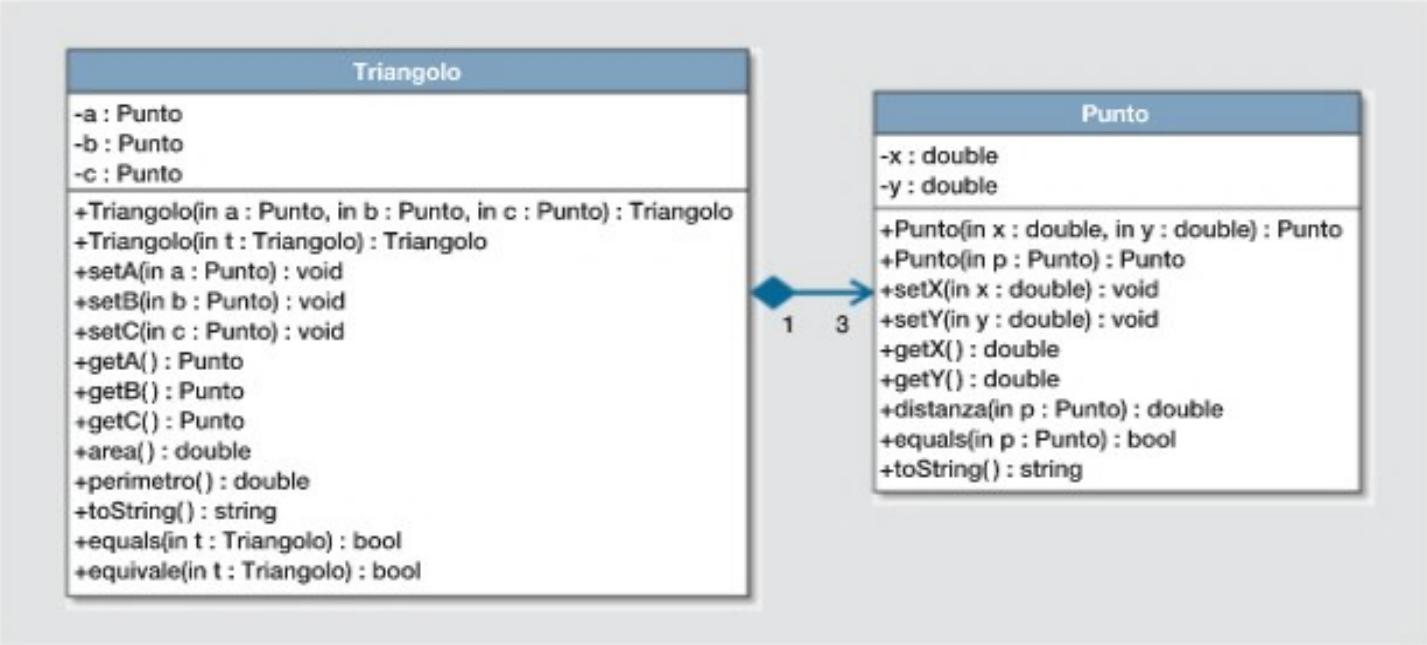


FIGURA **7**

Il codice che segue è una possibile implementazione della classe *Triangolo* in Java:

```
1   package geometria;
2
3   public class Triangolo {
4     private Punto a;
5     private Punto b;
6     private Punto c;
7
8     private final double ERR_MAX = 0.00000001;
9
10    public Triangolo (Punto a, Punto b, Punto c) {
11    setA(a);
12    setB(b);
13    setC(c);
14    }
15
16    public Triangolo (Triangolo t) {
17      setA(t.getA());
18      setB(t.getB());
19      setC(t.getC());
20  }
21
22    public void setA(Punto a){this.a = new Punto(a);}
```

```
23    public void setB(Punto b){this.b = new Punto(b);}
24    public void setC(Punto c){this.c = new Punto(c);}
25
26    public Punto getA(){return a;}
27    public Punto getB(){return b;}
28    public Punto getC(){return c;}
29
30    public double area() {
31      double p = perimetro()/2;
32      return Math.sqrt(p*(p-a.distanza(b))*(p-b.distanza(c))*
33      (p-c.distanza(a)));
34    }
35    public double perimetro() {
36      return a.distanza(b)+b.distanza(c)+c.distanza(a);
37    }
38
39    public String toString() {
40      String msg = "";
41      if (area() <= ERR_MAX) msg = " NON È UN TRIANGOLO!";
42      return " A"+a.toString()+" B"+b.toString()+
43        " C"+c.toString()+msg;
44    }
45
46    public boolean equals(Triangolo t) {
47      return ((a.equals(t.a))&&(b.equals(t.b))&&
              (c.equals(t.c)));
48    }
49
50    public boolean equivale(Triangolo t) {
51      return (Math.abs(this.area()-t.area()) <= ERR_MAX);
52    }
53
54    public static void main (String[] args) {
55      Punto p1 = new Punto(1.,1.);
56      Punto p2 = new Punto(2.,2.);
57      Punto p3 = new Punto(2.,1.);
58      Punto p4 = new Punto(1.,4.);
59      Punto p5 = new Punto(3.,3.);
60    Triangolo t1 = new Triangolo(p1,p2,p3);
61    Triangolo t2 = new Triangolo(p1,p3,p4);
62    Triangolo t3 = new Triangolo(p1,p2,p2);
63    Triangolo t4 = new Triangolo(t1);
64
65    System.out.println("Triangolo T1: "+t1.toString());
66    System.out.println("Perimetro triangolo T1:
      "+t1.perimetro());
67    System.out.println("Area triangolo T1: "+t1.area());
68    System.out.println("Triangolo T2: "+t2.toString());
69    System.out.println("Perimetro triangolo T2:
      "+t2.perimetro());
70    System.out.println("Area triangolo T2: "+t2.area());
71    System.out.println("Triangolo T3: "+t3.toString());
72    System.out.println("Perimetro triangolo T3:
      "+t3.perimetro());
```

```
73    System.out.println("Area triangolo T3: "+t3.area());
74    System.out.println("Triangolo T4: "+t4.toString());
75    System.out.println("Perimetro triangolo T4:
      "+t4.perimetro());
76    System.out.println("Area triangolo T4: "+t4.area());
77
78    if (t1.equals(t4))
79      System.out.println("I triangoli T1 e T4 sono uguali");
80    else
81      System.out.println("I triangoli T1 e T4 non sono
        uguali");
82    if (t1.equivale(t4))
83      System.out.println("I triangoli T1 e T4 sono
        equivalenti");
84    else
85      System.out.println("I triangoli T1 e T4 non sono
          equivalenti");
86
87    if (t1.equals(t2))
88      System.out.println("I triangoli T1 e T2 sono uguali");
89    else
90      System.out.println("I triangoli T1 e T2 non sono
        uguali");
91    if (t1.equivale(t2))
92      System.out.println("I triangoli T1 e T2 sono
        equivalenti");
93    else
94      System.out.println("I triangoli T1 e T2 non sono
        equivalenti");
95    }
96  }
```

che produce il seguente output:

```
Triangolo T1: A(1.0;1.0) B(2.0;2.0) C(2.0;1.0)
Perimetro triangolo T1: 3.414213562373095
Area triangolo T1: 0.4999999999999998
Triangolo T2: A(1.0;1.0) B(2.0;1.0) C(1.0;4.0)
Perimetro triangolo T2: 7.16227766016838
Area triangolo T2: 1.5000000000000007
Triangolo T3: A(1.0;1.0) B(2.0;2.0) C(2.0;2.0) NON È UN
TRIANGOLO!
Perimetro triangolo T3: 2.8284271247461903
Area triangolo T3: 0.0
Triangolo T4: A(1.0;1.0) B(2.0;2.0) C(2.0;1.0)
Perimetro triangolo T4: 3.414213562373095
Area triangolo T4: 0.4999999999999998
I triangoli T1 e T4 sono uguali
I triangoli T1 e T4 sono equivalenti
I triangoli T1 e T2 non sono uguali
I triangoli T1 e T2 non sono equivalenti
```

OSSERVAZIONE Volendo troncare i decimali di una variabile *v* (per esempio 2) si può usare *System.out.printf("%2f",v)*.

Analizziamo i dettagli del codice (tra parentesi sono indicati i numeri di riga a cui si fa riferimento).

- Si utilizza la classe *Punto* prima definita per rappresentare per ogni oggetto di classe *Triangolo* i tre punti che rappresentano i suoi vertici (4-6).
- Esistono due diversi costruttori in *overloading* (10-14, 16-20): il primo utilizza i tre metodi *setter* per istanziare un triangolo a partire dai tre vertici forniti come argomenti, il secondo è un costruttore di copia che istanzia un triangolo a partire da un diverso triangolo fornito come argomento di cui copia i vertici estratti tramite i tre metodi *getter*.
- Sono definiti metodi *getter* e *setter* per i tre attributi dichiarati di tipo privato (22-24, 26-28). Nello specifico, i tre metodi *setter* usano il costruttore di copia della classe *Punto* per creare i vertici del triangolo in fase di istanziamento come nuovi oggetti (in caso contrario, per esempio, in caso di copia di un triangolo, avremmo due triangoli con gli stessi oggetti come vertici: modificando il vertice di uno dei due triangoli si modificherebbe di conseguenza anche l'altro).
- Si è fatto ricorso al metodo statico *sqrt* della classe *Math* (32) per calcolare la radice quadrata necessaria nella formula di Erone per la determinazione dell'area del triangolo (se l'area è nulla i punti dei vertici di fatto coincidono e il triangolo è degenere).
- È stato definito un metodo per il calcolo del perimetro del triangolo (35-37) e ridefinito il metodo *equals* per la verifica dell'uguaglianza di due triangoli (46-48).
- È stato ridefinito il metodo *toString* per la restituzione in formato testuale del contenuto informativo di un oggetto di tipo *Triangolo* (39-44).

  OSSERVAZIONE L'operatore «+» è stato usato per concatenare più stringhe di caratteri in una sola stringa.

- Il metodo *equivale* (50-52) deve semplicemente verificare se le aree di due triangoli sono o meno uguali.

  OSSERVAZIONE I valori delle aree a causa degli inevitabili errori di approssimazione numerica presenti nei risultati dei calcoli con numeri non interi non sono confrontati direttamente, cosa che porterebbe a considerare diversi valori di fatto uguali ma affetti da piccoli errori di approssimazione. Si verifica che la loro differenza in valore assoluto – determinato invocando il metodo statico *abs* della classe *Math* – sia minore di un errore massimo prefissato, in questo caso dato dall'attributo costante ERR_MAX.

- È stato definito il metodo *main* per la verifica delle funzionalità dei metodi della classe (54-95).

5 Convenzioni di codifica del linguaggio Java

Nella codifica di programmi in linguaggio Java ci sono alcune convenzioni da seguire per lo stile del codice sorgente. Il linguaggio Java – così come il linguaggio C/C++ – è *case-sensitive* (distingue cioè i caratteri maiuscoli da quelli minuscoli) per cui è necessario fare attenzione a come sono scritte sia le parole chiave sia gli identificatori.

Per quanto riguarda gli identificatori vengono utilizzate convenzioni che, benché irrilevanti ai fini della compilazione, sono invece importanti per la leggibilità del codice:

- i nomi delle classi devono iniziare con una lettera maiuscola;
- i nomi di metodi, attributi e variabili iniziano con una lettera minuscola;
- i nomi composti usano la convenzione *LowerCamelCase*: le parole vengono riportate in minuscolo, una di seguito all'altra senza caratteri di separazione, usando un carattere maiuscolo come lettera iniziale di ogni parola, esclusa la prima se non si tratta di un nome di classe.

Inoltre, tali nomi dovrebbero essere sempre scelti in modo da:

- descrivere adeguatamente ciò che identificano (esempio: *FiguraGeometrica* come nome di una classe, o *getDescrizione* come nome di un metodo che restituisce la descrizione di un oggetto);
- essere pronunciabili e formati da parole complete piuttosto che da abbreviazioni (esempio: *valoreMedioRelativo* piuttosto che *valMedRel*);
- essere di lunghezza contenuta, ma sufficientemente descrittivi (esempio: *ElencoArticoli* e non *ElencoArticoliAcquistabiliSuWeb*);
- essere omogenei in tutto il codice oggetto di sviluppo (esempio: se si è utilizzato il termine *Prodotto* come nome di classe è preferibile definire nella classe *Ordini* il metodo *aggiungiProdotto* piuttosto che *aggiungiArticolo*);
- esprimere affermatività (esempio: è preferibile per il nome di una variabile booleana *trovato* piuttosto che *nonTrovato*).

Infine, ogni riga non dovrebbe essere troppo lunga (normalmente non più di 80 caratteri) per evitare lo scorrimento orizzontale del testo che richiede tempo e affatica la lettura: è spesso possibile suddividere un'istruzione su più righe di codice.

Le convenzioni per i componenti Java riusabili (*Javabeans*) prevedono che per ogni attributo vi siano due specifici metodi, che consentono di acquisirne o impostarne il valore avendo come nome rispettivamente i prefissi *get* o *set* seguiti dal nome dell'attributo:

```
public String getDatiUtente() {...};
public void setDatiUtente(String datiUtente) {...};
```

Nel caso di un valore da restituire di tipo booleano, il prefisso da utilizzare non è *get*, ma *is*

```
public boolean isFunzionante() {...};
```

oppure *has* se più indicato come verbo:

```
public boolean hasDipendenti() {...};
```

OSSERVAZIONE Molti IDE generano automaticamente i metodi *get/set* e il costruttore a partire dalla definizione degli attributi di una classe.

Per quanto riguarda il codice dei metodi si dovrebbero osservare le seguenti regole.

- Per i blocchi di codice le parentesi si aprono al termine della riga dell'istruzione e si chiudono in corrispondenza della colonna di inizio dell'istruzione stessa.

ESEMPIO

```
if (n == 0) {
   ...
   ...
   ...
}

while (n > 0) {
   ...
   ...
   ...
}

do {
   ...
   ...
   ...
} while(!fine);
```

- L'indentazione del codice nel corpo delle istruzioni `if-else`, `switch-case`, `while`, `do-while` e `for` deve essere fissata in un numero costante di spazi (normalmente gli 8 spazi di una tabulazione) in modo da evidenziare chiaramente la struttura del codice.
- Ogni riga non deve contenere più istruzioni separate dal simbolo «;».

6 Tipi di dato primitivi e classi *wrapper*

Java è un linguaggio staticamente tipizzato: questo significa che ogni volta che dichiariamo una variabile dobbiamo specificare il tipo della variabile, che può essere uno dei tipi primitivi del linguaggio, oppure una classe predefinita, della libreria, o definita dall'utente. La TABELLA 2 elenca tutti i tipi primitivi del linguaggio Java.

TABELLA 2

int	Il tipo **int** rappresenta un numero intero a 32 bit che può assumere un valore compreso tra: −2.147.483.648 e +2.147.483.647 Il valore di default che assume una variabile non inizializzata è 0.
long	Il tipo **long** rappresenta un numero intero a 64 bit e può assumere un valore compreso tra: −9.223.372.036.854.775.808 e +9.223.372.036.854.775.807 Il valore di default che assume una variabile non inizializzata è 0l; il suffisso «l» indica il tipo **long**.
short	Il tipo **short** rappresenta un numero intero a 16 bit e può assumere un valore compreso tra: −32,768 e +32,767 Il valore di default che assume una variabile non inizializzata è 0.
float	Il tipo **float** rappresenta un numero in virgola mobile (*floating-point*) a 32 bit secondo lo standard IEEE-754 e può assumere un valore reale compreso tra $+/-2^{-149}$ (circa 1.04239846E−45) e $+/-(2.2^{-23}) \times 2^{127}$ (circa 3.40282347E+38) e ha 7 cifre decimali significative. Il valore di default che assume una variabile non inizializzata è 0.0f, dove il suffisso «f» indica il tipo **float**.
double	Il tipo **double** rappresenta un numero in virgola mobile (*floating-point*) a 64 bit secondo lo standard IEEE-754 e può assumere un valore compreso tra $+/-2^{-1074}$ (circa 4.94065645841246544E−324) e $+/-(2.2^{-52}) \times 2^{1023}$ (circa 1.79769313486231570E+38) e ha 15 cifre decimali significative. Il valore di default che assume una variabile non inizializzata è 0.0.
byte	Il tipo **byte** rappresenta un numero intero compreso tra −128 e +127. Il valore di default che assume una variabile non inizializzata è 0.
boolean	Il tipo **boolean** può assumere i soli valori **true** e **false**. Il valore di default che assume una variabile non inizializzata è **false**.
char	Il tipo **char** rappresenta un singolo carattere Unicode a 16 bit: i valori rappresentati si estendono da '\u0000' (0) a '\uFFFF' (65535) Il valore di default che assume una variabile non inizializzata è '\u0000'.

In molte situazioni può tornare utile trattare i dati primitivi come oggetti: a questo proposito il linguaggio Java rende disponibili le classi denominate *wrapper* (o *adapter*) nel *package java.lang*, che non richiede di essere importato. Una classe *wrapper* incapsula una variabile di tipo primitivo, ovvero trasforma un tipo primitivo di cui mantiene il valore in un oggetto corrispondente che integra alcune utili funzionalità. Nella maggior parte dei casi la classe *wrapper* ha lo stesso nome del tipo primitivo corrispondente come mostra la TABELLA 3.

TABELLA 3

Tipo primitivo	Classe *wrapper*
byte	`Byte`
boolean	`Boolean`
char	`Character`
int	`Integer`
long	`Long`
short	`Short`
float	`Float`
double	`Double`

OSSERVAZIONE La sintassi utilizzata per dichiarare gli oggetti di tipo *String* nell'esempio precedente pare quella utilizzata per i tipi primitivi, ma in realtà, trattandosi di un oggetto, a tutti gli effetti una variabile di tipo *String* come *messaggio*, è un riferimento a un oggetto istanza della classe *String* creato nello *heap* e il cui valore è *Hello, world!*. In effetti è possibile usare una sintassi più tradizionale per dichiarare e creare l'oggetto *messaggio*:

```
String messaggio = new String("Hello, world!");
```

Le sintassi utilizzate nell'esempio consentono al programmatore di scrivere codice in modo più naturale e veloce, rendendo implicita l'invocazione dell'operatore **new**.

Una stringa può contenere un numero qualsiasi di caratteri; la stringa vuota non ne contiene nessuno ed è rappresentata dal simbolo «""».

OSSERVAZIONE La notazione «" "» è diversa da «""»: la prima stringa contiene uno spazio bianco, mentre la seconda è vuota.

Diversamente da altri linguaggi di programmazione come C/C++, la codifica dei caratteri in Java non è basata sullo standard ASCII, la cui codifica numerica è limitata a un byte e quindi ai soli 256 simboli alfanumerici che si possono codificare con 8 bit, ma utilizza il set di caratteri Unicode, la cui codifica numerica impiega 16 bit con i quali è possibile codificare fino a 65 536 simboli diversi. I progettisti del linguaggio Java hanno adottato il set di caratteri Unicode per supportare in modo adeguato i vari alfabeti internazionali. Dato che i primi 256 simboli del set Unicode sono i simboli dello standard ASCII, l'adozione di questa codifica non ha alcuna conseguenza per le applicazioni che utilizzano l'alfabeto occidentale.

L'operatore «+» usato con variabili di tipo stringa consente di concatenare tra loro due stringhe per crearne una nuova il cui contenuto è dato dalla sequenza di caratteri della seconda stringa giustapposti alla sequenza di caratteri della prima stringa (l'operatore «+» può essere utilizzato associativamente per concatenare più di due stringhe).

ESEMPIO Il seguente frammento di codice posto in un eventuale metodo *main* di una classe Java

```
...
String saluto = "Hello, ";
String mondo = "world!");
saluto = saluto+mondo;
System.out.println(saluto);
...
```

visualizzerebbe il seguente messaggio:

```
Hello, world!
```

OSSERVAZIONE La sintassi utilizzata per dichiarare gli oggetti di tipo *String* nell'esempio precedente pare quella utilizzata per i tipi primitivi, ma in realtà, trattandosi di un oggetto, a tutti gli effetti una variabile di tipo *String* come *messaggio*, è un riferimento a un oggetto istanza della classe *String* creato nello *heap* e il cui valore è *Hello, world!*. In effetti è possibile usare una sintassi più tradizionale per dichiarare e creare l'oggetto *messaggio*:

```
String messaggio = new String("Hello, world!");
```

Le sintassi utilizzate nell'esempio consentono al programmatore di scrivere codice in modo più naturale e veloce, rendendo implicita l'invocazione dell'operatore `new`.

Una stringa può contenere un numero qualsiasi di caratteri; la stringa vuota non ne contiene nessuno ed è rappresentata dal simbolo «""».

OSSERVAZIONE La notazione «" "» è diversa da «""»: la prima stringa contiene uno spazio bianco, mentre la seconda è vuota.

Diversamente da altri linguaggi di programmazione come C/C++, la codifica dei caratteri in Java non è basata sullo standard ASCII, la cui codifica numerica è limitata a un byte e quindi ai soli 256 simboli alfanumerici che si possono codificare con 8 bit, ma utilizza il set di caratteri Unicode, la cui codifica numerica impiega 16 bit con i quali è possibile codificare fino a 65 536 simboli diversi. I progettisti del linguaggio Java hanno adottato il set di caratteri Unicode per supportare in modo adeguato i vari alfabeti internazionali. Dato che i primi 256 simboli del set Unicode sono i simboli dello standard ASCII, l'adozione di questa codifica non ha alcuna conseguenza per le applicazioni che utilizzano l'alfabeto occidentale.

L'operatore «+» usato con variabili di tipo stringa consente di concatenare tra loro due stringhe per crearne una nuova il cui contenuto è dato dalla sequenza di caratteri della seconda stringa giustapposti alla sequenza di caratteri della prima stringa (l'operatore «+» può essere utilizzato associativamente per concatenare più di due stringhe).

ESEMPIO Il seguente frammento di codice posto in un eventuale metodo *main* di una classe Java

```
...
String saluto = "Hello, ";
String mondo = "world!");
saluto = saluto+mondo;
System.out.println(saluto);
...
```

visualizzerebbe il seguente messaggio:

```
Hello, world!
```

OSSERVAZIONE In Java gli oggetti di tipo *String* sono «immutabili», cioè costanti: assumono un valore al momento della creazione che non può successivamente più essere modificato. Di conseguenza «modificare» una stringa equivale a distruggere l'oggetto esistente e crearne uno nuovo. L'esecuzione del codice dell'esempio precedente comporta i seguenti passi:

- creazione della variabile *saluto* che riferisce nello *heap* un oggetto di tipo stringa con valore *Hello*;
- creazione della variabile *mondo* che riferisce nello *heap* un oggetto di tipo stringa con valore *world*;
- creazione di un oggetto nello *heap* di tipo stringa il cui valore è *Hello, world!*;
- cambiamento del riferimento della variabile *saluto* al nuovo oggetto di tipo stringa creato;
- distruzione dell'oggetto di tipo stringa con valore *Hello* precedentemente riferito dalla variabile *saluto* e ora non più riferito da parte del *garbage collector* che ne recupera la memoria occupata.

In questo lungo procedimento nessuna stringa ha cambiato valore!

OSSERVAZIONE L'uso dell'operatore «+» richiede un po' di attenzione perché, essendo polimorfico, in alcuni casi può portare a risultati non voluti. Le seguenti istruzioni Java

```
System.out.println("numero di telefono "+1+5+":"+339+1116798);
System.out.println(339+1116798+" numero di telefono "+1+5);
```

producono rispettivamente il seguente output:

```
numero di telefono 15: 3391116798
```

```
1117137 numero di telefono 15
```

Nel primo caso l'operatore «+» viene interpretato come operatore di concatenazione tra stringhe, essendo il primo operando una stringa (`"numero di telefono "`): gli operandi numerici che seguono sono convertiti automaticamente in una stringa prima di effettuare la concatenazione. Nel secondo caso i primi due operandi sono numerici, per cui il simbolo «+» viene interpretato come operatore algebrico e prima di eseguire la concatenazione con il terzo operando, che richiede la conversione in stringa, viene effettuata la somma; gli operandi successivi, pur essendo valori numerici, sono ormai interpretati come stringhe, perché uno dei due operandi è una stringa. Per ottenere il risultato voluto anche nel secondo caso si sarebbe potuto scrivere così:

```
System.out.println(""+339+1116798+" numero di telefono "+1+5);
```

Essendo il primo operando una stringa vuota, l'operatore sarebbe stato interpretato come operatore di concatenazione e, convertendo i valori numerici in stringhe, avrebbe prodotto il seguente output:

```
3391116798 numero di telefono 15
```

La classe *String* rende disponibili molti metodi per la gestione delle stringhe di caratteri di cui la TABELLA 7 riporta i più significativi.

TABELLA 7

Metodo	Descrizione
`s.length()`	Restituisce la lunghezza in caratteri della stringa *s*.
`s.charAt(ind)`	Restituisce il carattere che si torva nella posizione *ind* della stringa *s*, tenendo conto che il primo carattere occupa la posizione 0; non è possibile utilizzare la notazione C/C++ `s[indice]` perché le stringhe in Java non sono *array* di caratteri e l'operatore di indicizzazione non può essere ridefinito.
`s.indexOf('x')`	Ritorna la posizione della prima occorrenza del carattere `'x'` nella stringa *s*, o il valore −1 se non è presente.
`s1.equals(s2)` `s1.equalsIgnoreCase(s2)`	Restituisce **`true`** se le stringhe *s1* ed *s2* hanno lo stesso contenuto, **`false`** altrimenti; non è possibile utilizzare l'operatore «==» per effettuare questo confronto perché verificherebbe se quanto riferito da *s1* ed *s2* è lo stesso oggetto, restituendo **`false`** in caso contrario anche se il contenuto degli oggetti riferiti è esattamente lo stesso. Il metodo *equalsIgnoreCase* ignora la differenza tra lettera maiuscola e minuscola.
`s2 = s1.substring(2,8)`	La stringa *s2* assume il valore della sottostringa di *s1* compresa tra i caratteri di indice 2 e 8.
`s2 = s1.replace('A','a')`	Restituisce in *s2* il valore della stringa *s1* in cui tutti i caratteri «A» sono stati sostituiti con caratteri «a».

OSSERVAZIONE Per confrontare l'uguaglianza di due stringhe di caratteri in Java – o, in generale, di due oggetti – è necessario utilizzare il metodo *equals* invece dell'operatore di confronto «==». Prendiamo in esame i due seguenti frammenti di codice:

```
String s1 = new String("alfa");
String s2 = new String("alfa");
System.out.println((s1==s2));

String s1 = "beta";
String s2 = "beta";
System.out.println((s1==s2));
```

Inaspettatamente nel primo caso sarà prodotto il risultato **`false`**, mentre nel secondo il risultato **`true`**. Il primo risultato è corretto perché il confronto avviene tra due riferimenti a stringhe diverse che hanno lo stesso valore (FIGURA 8).
Come mai il secondo caso fornisce un risultato diverso? Nel primo caso il ricorso esplicito all'operatore **`new`** crea due distinti oggetti di tipo stringa nello *heap* che l'operatore «==» considera diversi. Nel secondo caso, dato che le due costanti di tipo stringa sono uguali, il compilatore Java effettua un'ottimizzazione creando una sola stringa nello *heap* (FIGURA 9).

La classe *String* rende disponibili molti metodi per la gestione delle stringhe di caratteri di cui la TABELLA 7 riporta i più significativi.

TABELLA 7

Metodo	Descrizione
`s.length()`	Restituisce la lunghezza in caratteri della stringa *s*.
`s.charAt(ind)`	Restituisce il carattere che si torva nella posizione *ind* della stringa *s*, tenendo conto che il primo carattere occupa la posizione 0; non è possibile utilizzare la notazione C/C++ `s[indice]` perché le stringhe in Java non sono *array* di caratteri e l'operatore di indicizzazione non può essere ridefinito.
`s.indexOf('x')`	Ritorna la posizione della prima occorrenza del carattere `'x'` nella stringa *s*, o il valore −1 se non è presente.
`s1.equals(s2)` `s1.equalsIgnoreCase(s2)`	Restituisce **`true`** se le stringhe *s1* ed *s2* hanno lo stesso contenuto, **`false`** altrimenti; non è possibile utilizzare l'operatore «==» per effettuare questo confronto perché verificherebbe se quanto riferito da *s1* ed *s2* è lo stesso oggetto, restituendo **`false`** in caso contrario anche se il contenuto degli oggetti riferiti è esattamente lo stesso. Il metodo *equalsIgnoreCase* ignora la differenza tra lettera maiuscola e minuscola.
`s2 = s1.substring(2,8)`	La stringa *s2* assume il valore della sottostringa di *s1* compresa tra i caratteri di indice 2 e 8.
`s2 = s1.replace('A','a')`	Restituisce in *s2* il valore della stringa *s1* in cui tutti i caratteri «A» sono stati sostituiti con caratteri «a».

OSSERVAZIONE Per confrontare l'uguaglianza di due stringhe di caratteri in Java – o, in generale, di due oggetti – è necessario utilizzare il metodo *equals* invece dell'operatore di confronto «==». Prendiamo in esame i due seguenti frammenti di codice:

```
String s1 = new String("alfa");
String s2 = new String("alfa");
System.out.println((s1==s2));

String s1 = "beta";
String s2 = "beta";
System.out.println((s1==s2));
```

Inaspettatamente nel primo caso sarà prodotto il risultato **`false`**, mentre nel secondo il risultato **`true`**. Il primo risultato è corretto perché il confronto avviene tra due riferimenti a stringhe diverse che hanno lo stesso valore (FIGURA 8).
Come mai il secondo caso fornisce un risultato diverso? Nel primo caso il ricorso esplicito all'operatore **`new`** crea due distinti oggetti di tipo stringa nello *heap* che l'operatore «==» considera diversi. Nel secondo caso, dato che le due costanti di tipo stringa sono uguali, il compilatore Java effettua un'ottimizzazione creando una sola stringa nello *heap* (FIGURA 9).

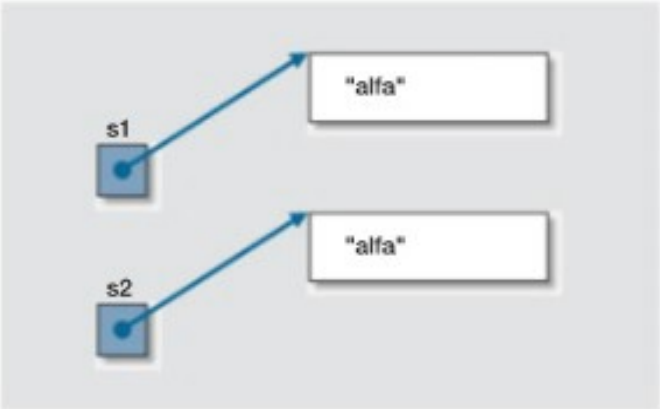


FIGURA 8

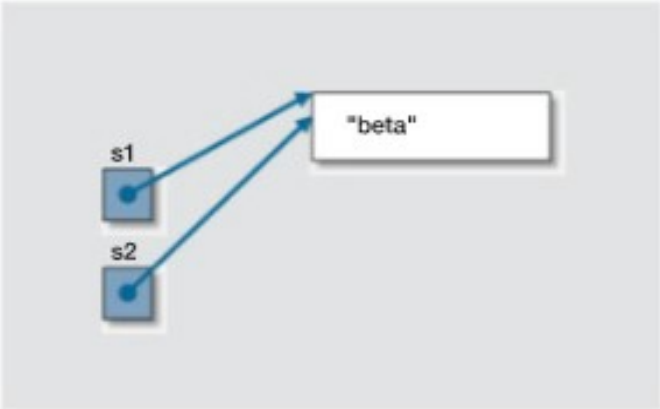


FIGURA 9

In pratica è come se il secondo frammento di codice fosse

```
String s1 = new String("beta");
String s2 = s1;
```

per cui i due riferimenti *s1* ed *s2* risultano uguali in quanto riferiscono lo stesso oggetto.

Gli oggetti di tipo *String* sono immutabili; ci sono però situazioni in cui risulta necessario modificare il contenuto di una stringa: in questi casi è conveniente istanziare oggetti della classe *StringBuffer* che fornisce, tra i molti altri, un metodo che consente di modificare singoli caratteri di una stringa.

ESEMPIO

Le seguenti istruzioni

```
StringBuffer s = new StringBuffer("alfa");
s.setCharAt(2,'b');
```

creano un riferimento a un oggetto di classe *StringBuffer* che contiene la stringa *alfa* e successivamente modificano il carattere di indice 2 (terza posizione) della stringa con un carattere *b* in modo che la nuova stringa sia *alba*. È possibile assegnare il contenuto di un oggetto di tipo *StringBuffer* a un oggetto di tipo *String* invocando il metodo *toString*. Dovendo modificare un oggetto di tipo *String* è possibile operare come di seguito:

```
String s = "alfa";
StringBuffer sb = new StringBuffer(s);
sb.setCharAt(2,'b');
s = sb.toString();
```

7.1 Stringhe e istruzione switch-case

A partire dalla versione Java 7 è possibile usare un oggetto di tipo *String* nell'uso dell'istruzione **switch-case**:

```
public String getTipoDiGiorno(String giornoSettimana) {
  String tipoDiGiorno;
  switch (giornoSettimana) {
    case "Lunedì":
      tipoDiGiorno = "Inizio settimana lavorativa";
      break;
    case "Martedì":
```

```
    case "Mercoledì":
    case "Giovedì":
      tipoDiGiorno = "Metà settimana lavorativa";
      break;
    case "Venerdì":
      tipoDiGiorno = "Fine settimana lavorativa";
      break;
    case "Sabato":
    case "Domenica":
      tipoDiGiorno = "Weekend";
      break;
    default:
      throw new IllegalArgumentException("Giorno settimana
      errato: " + giornoSettimana);
  }
  return tipoDiGiorno;
}
```

L'istruzione `switch-case` confronta l'agomento *String* con l'espressione associata a ogni `case` usando il metodo *equals* (confronto *case-sensitive*). Il compilatore Java genera in questo modo un codice più efficiente rispetto a un'equivalente catena di istruzioni `if-then-else`.

Nella sezione `default` dell'esempio viene rilevato un errore e sollevata (`throw`) un'eccezione (*IllegalArgumentException*), ovvero un meccanismo le cui caratteristiche saranno trattate nei prossimi capitoli.

8 Gestione delle date e degli orari

Una delle necessità comuni a molte applicazioni è quella relativa alla gestione di date e orari. Nelle versioni di Java precedenti la 8 erano disponibili le classi *java.util.Date* e *java.util.Calendar* che presentavano diversi tipi di problemi tra cui:

- creazione oggetti *data* mutabili (una data dovrebbe essere auspicabilmente costante);
- rappresentazione di intervalli di tempo (espressi in millisecondi dal 1 gennaio 1970 al momento di creazione di una data);
- inadeguatezza (*thread-unsafe*), salvo adeguati accorgimenti, per applicazioni concorrenti;
- mesi numerati partendo da 0, con conseguente possibile confusione;
- le classi di utilità per formattare le date utilizzabili solo con *Date* ma non con *Calendar*.

Questo stato di cose ha incentivato la creazione di API di terze parti per la loro soluzione. Un esempio in tal senso è *Joda-Time* a cui, successivamente, Java 8 si è ispirato per la creazione di una nuova soluzione proprietaria.

La nuova API è stata sviluppata in modo da offrire classi:

- di oggetti immutabili che rappresentano un valore e non un intervallo temporale;

- *thread-safe* che gestiscono automaticamente gli aspetti relativi alla concorrenza;
- con caratteristiche *domain-driven design* per tener conto delle possibili situazioni in cui date e orari vengono utilizzati e offrendo al tempo stesso un loro uso immediato e comprensibile;
- con disponibilità di diversi sistemi di calendario, per supportare le esigenze di particolari zone geografiche (per esempio in Asia talvolta non viene utilizzato lo standard ISO-8601 relativo al calendario gregoriano).

Essa prevede i seguenti *package*:

- *java.time*, contiene le classi base (sufficienti nella maggior parte delle esigenze);
- *java.time.chrono*, permette l'accesso a differenti tipi di calendari;
- *java.time.format*, contiene le classi per la formattazione e il *parsing* di date e orari;
- *java.time.temporal*, estende il *package* base mettendo a disposizione classi per la manipolazione più a basso livello di date e orari;
- *java.time.zone*, contiene le classi per la gestione delle *time zone* geografiche.

8.1 Gestione delle date

Per la gesione delle date la classe più importante è *LocalDate*. Questa classe permette la rappresentazione di una data senza alcuna informazione oraria (per esempio «6 ottobre 2016») che avrà inizio in momenti diversi in base alla sua posizione sulla terra (per esempio a New York la stessa data locale inizierà 6 ore dopo che a Roma).

Quando si costruisce una data partendo dal valore dei suoi componenti (giorno, mese, anno) il nome del costruttore (*factory method*) è *of*, mentre se si crea una data a partire da un'altra data (costruttore di copia) si usa il metodo *from*. È inoltre previsto un opportuno costruttore per istanziare una data fornendo come argomento una stringa.

Di seguito, a titolo di esempio, vediamo come viene istanziata una data e successivamente elaborata per estrarre da essa alcune informazioni.

```
LocalDate data = LocalDate.of(2015, Month.SEPTEMBER, 8);
int anno = data.getYear(); // 2015
Month mese = data.getMonth(); // SEPTEMBER
int giornoDelMese = data.getDayOfMonth(); // 8
DayOfWeek giornoDellaSettimana = data.getDayOfWeek();
// Martedì
int numeroGiorniMese = data.lengthOfMonth();
// 30 (Numero di gg di settembre)
boolean bisestile = data.isLeapYear();
// false (2015 non è bisestile)
```

Ulteriori tipi di elaborazione sono quelli seguenti:

```
LocalDate data = LocalDate.of(2016, Month.SEPTEMBER, 10);
LocalDate data2 = data.withYear(2015); // 10-09-2015
data2 = data.plusMonths(2); // 10-11-2016
data2 = data.minusDays(1); // 09-09-2016
```

Per elaborazioni particolari, il *package java.time.temporal* (in particolare la classe *TemporalAdjuster*) offre delle utilità predefinite attraverso vari metodi statici, detti *adjuster*, che permettono manipolazioni standard sulle date. In caso di bisogno, è comunque possibile implementare *adjuster* personalizzati. Per esempio:

```
import static java.time.DayOfWeek.*
import static java.time.temporal.TemporalAdjusters.*

LocalDate data = LocalDate.of(2016, Month.SEPTEMBER, 10);
data = data.with(lastDayOfMonth());
data = data.with(nextOrSame(WEDNESDAY));
```

8.2 Gestione degli orari

La classe *LocalTime* è l'omologa di *LocalDate* per la gestione degli orari. *LocalTime* rappresenta un valore orario senza alcuna data associata (né fuso orario) ed è relativo a un determinato fuso orario.

```
LocalTime tempo = LocalTime.of(20, 30);
int ora = tempo.getHour(); // 20
int minuto = tempo.getMinute(); // 30
tempo = tempo.withSecond(6); // 20:30:06
tempo = tempo.plusMinutes(3); // 20:33:06
```

Anche in questo caso sono previsti specifici *adjuster*.

8.3 Gestione di data e ora congiuntamente

È possibile combinare data e ora con la classe *LocalDateTime*:

```
LocalDateTime dt1 = LocalDateTime.of(2014, Month.JUNE, 10, 20, 30);
LocalDateTime dt2 = LocalDateTime.of(data, tempo);
LocalDateTime dt3 = data.atTime(20, 30);
LocalDateTime dt4 = data.atTime(tempo);
```

Nell'esempio precedente, il metodo *atTime*, serve a creare un oggetto *date-time* combinando una data con un orario.

La maggior parte delle classi presenti nella API espongono metodi «*at*» per la creazione di oggetti complessi unendo più oggetti *Date* e/o *Time*. Gli altri metodi di *LocalDateTime* sono simili a quelli di *LocalDate* e *LocalTime*.

Di seguito, una tabella riassuntiva dei metodi previsti dalla API e il loro significato.

Metodo	Descrizione
`of`	Costruttore per la creazione di un oggetto a partire dalle sue componenti.
`from`	Costruttore che crea un'istanza a partire da un oggetto dello stesso tipo.
`now`	Costruttore che crea un'istanza orario impostata all'ora corrente.
`parse`	Costruttore che crea un'istanza sulla base di una stringa passata in input.
`get`	Restituisce una parte (giorno, mese, ...) di un oggetto *date-time*.
`is`	Verifica se una qualche proprietà di un oggetto *date-time* è vera o falsa.
`with`	Restituisce una copia di un oggetto *date-time* con qualche proprietà modificata.
`plus`	Restituisce una copia di un oggetto *date-time* con un valore di una qualche proprietà aumentata di una certa quantità di tempo. ▶

Metodo	Descrizione
`minus`	Restituisce una copia di un oggetto *date-time* con un valore di una qualche proprietà diminuita di una certa quantità di tempo.
`to`	Converte un oggetto *date-time* in un altro oggetto.
`at`	Combina un oggetto *date-time* con altri oggetti per creare un'altro oggetto *date-time* più complesso.
`format`	Formatta un oggetto *date-time* come stringa nel formato desiderato.

Gli oggetti di tipo *date-time* sono immutabili come gli oggetti di tipo *String* e quindi anche essi possono essere confrontati tra di loro utilizzando il metodo *equals* piuttosto che l'operatore «==».

8.4 Il nuovo sistema dei calendari

La classe *LocalDate*, così come la maggior parte delle classi dell'API *java.time*, usa un unico calendario e, in particolare, lo standard ISO-8601 che è relativo al calendario gregoriano (sistema più usato al mondo). Al fine di supportare gli sviluppatori che utilizzano dei sistemi non ISO in applicazioni fortemente localizzate, l'API *java.time* offre un meccanismo di gestione di vari tipi di calendario. L'interfaccia *Chronology* è la classe base su cui è costruito il sistema di gestione dei calendari. Oltre al calendario gregoriano vi sono quelli: buddista thailandese, Minguo, giapponese e Hira (classi *ThaiBuddhistDate*, *MinguoDate*, *JapaneseDate* e *HijrahDate*).

Sono infine previste anche altre importanti interfacce che permettono di creare sistemi di calendario personalizzati, in particolare: *ChronoLocalDate*, *ChronoLocalDateTime*, *ChronoZonedDateTime* (utilizzate quasi esclusivamente da sviluppatori su applicazioni altamente internazionalizzate per tener conto dei sistemi di calendario locali).

8.5 La classe *Instant*

Quando si opera con date e orari in genere si pensa, secondo un modello di tempo «umano», in termini di anni, mesi, giorni, ore, minuti e secondi. Tuttavia un altro modello di uso comune è il tempo «macchina» (o «continuo») simile a quello della «vecchia» classe *Date*.

L'API *java.time* fornisce una visione del tempo macchina tramite il tipo di valore istantaneo. Esso offre la possibilità di rappresentare un punto sulla scala temporale senza altre informazioni contestuali, come un fuso orario e rappresenta il numero di secondi trascorsi dalla mezzanotte del 1 gennaio 1970 UTC. La classe *Instant* fornisce una precisione nell'ordine dei nanosecondi.

Per esempio:

```
Instant inizio = Instant.now();
// qualche calcolo
Instant fine = Instant.now();
assert fine.isAfter(inizio);
```

La classe *Instant* viene generalmente utilizzata per la memorizzazione e il confronto di *timestamp*, in cui è necessario registrare quando si è verificato un certo evento, tuttavia per quanto riguarda questa classe è meglio soffer-

marsi su cosa non si può fare piuttosto che su cosa si può fare. Per esempio, le righe di codice che seguono sollevano un'eccezione perché *Instant* rappresenta solo un numero di secondi e nanosecondi senza «consapevolezza» delle unità temporali «umane» quali possono essere mesi o anni:

```
instant.get(ChronoField.MONTH_OF_YEAR);
instant.plus(6, ChronoUnit.YEARS);
```

9 Java Shell

JShell, introdotto con Java 9, è un ambiente interattivo per l'esecuzione a riga di comando di espressioni Java. Molti linguaggi compilati includono *tool* di questo tipo (talvolta chiamati REPL, *Read Eval Print Loop*) che permettono di testare piccole parti di codice velocemente senza dover creare un progetto.

La cartella *bin* della versione Java 9 contiene l'eseguibile *jshell*. Digitandone il nome si accede al relativo ambiente che risponde con il *prompt* **`jshell>`** (è possibile lanciare *jShell* in modalità verbose specificando il parametro *-v* e ritornare allo standard utilizzando `"/set feedback normal"`).

```
jdk-9/bin/jshell -v
|   Welcome to JShell -- Version 9-ea
|   For an introduction type: /help intro

jshell>
```

L'uscita da *JShell* è resa possibile dal comando `/exit`.

Per default si può invocare un insieme base di librerie da importare a cui ne possono essere aggiunti altri in qualsiasi momento:

```
jshell> /imports
|    import java.util.*
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.util.concurrent.*
|    import java.util.prefs.*
|    import java.util.regex.*
```

Tra l'altro è possibile:

- digitare una qualsiasi espressione Java che, una volta valutata, restituisce un valore (assegnandolo a una variabile) e il suo tipo:

  ```
  jshell> 3+3
  $1 ==> 6
  |  created scratch variable $1 : int

  jshell> $1
  $1 ==> 6
  |  value of $1 : int
  ```

- dichiarare una variabile specificandone il nome in modo da renderla visibile:

  ```
  jshell> int x=5
  ```

```
x ==> 5
|  created variable x : int

jshell> x
x ==> 5
|  value of x : int
```

- definire metodi e magari modificarli successivamente:

```
jshell> void helloJShell() { System.out.println "Ciao JShell"); }
|  created method helloJShell()

jshell> helloJShell();
Ciao JShell

jshell> void helloJShell() { System.out.println ("Ho riscritto un metodo"); }
|  modified method helloJShell()
|    update overwrote method helloJShell()

jshell> helloJShell()
Ho riscritto un metodo
```

- verificare velocemente i tipi ritornati da alcune operazioni:

```
jshell> 1/2
$1 ==> 0
|  created scratch variable $1 : int

jshell> 1.0/2
$2 ==> 0.5
|  created scratch variable $2 : double

jshell> 1/2.0f
$5 ==> 0.5
|  created scratch variable $5 : float
```

Talvolta i messaggi hanno bisogno di essere correttamente formattati. Questo è un compito noioso che spesso porta a cicli di ricompilazione che rallentano il lavoro in modo significativo. Immaginiamo, per esempio, di aver dimenticato quale fosse il segno di formattazione per gli interi. È possibile verificare rapidamente che:

provando con «%i» (int) si verifica un errore

```
jshell> printf("Ho preso %i mela",1)
|   java.util.UnknownFormatConversionException thrown: Conversion = 'i'
|     at Formatter$FormatSpecifier.conversion (Formatter.java:2691)
|     at Formatter$FormatSpecifier.<init> (Formatter.java:2717)
|     at Formatter.parse (Formatter.java:2565)
|     at Formatter.format (Formatter.java:2507)
|     at PrintStream.format (PrintStream.java:977)
```

```
|      at PrintStream.printf (PrintStream.java:873)
|      at printf (#s8:1)
|      at (#51:1)
```

mentre con «%d» (decimal) la soluzione è corretta:

```
jshell> printf("Ho preso %d mela",1)
Ho preso 1 mela
```

JShell prevede un insieme di comandi i cui nomi iniziano con «/» e la cui lista completa è ottenibile `/help`. Alcuni dei più usati sono quelli relativi alla verifica dello stato corrente dell'ambiente o all'uso di un editor esterno per modificare i metodi utilizzati (TABELLA **8**).

TABELLA **8**

Comando	Effetto	Esempi

`/vars`	Elenco delle variabili	`jshell> /vars`
		`|  int x = 0`
		`|  double j = 0.5`

`/methods`	Elenco dei metodi attivi	`jshell> /methods`
		`|  printf (String,Object...)void`
		`|  helloJShell ()void`

`/list`	Elenco del codice dei metodi	`jshell> /list`
		`14 : helloJShell();`
		`15 : void helloJShell() {System.out.println("Ho riscritto un metodo");}`
		`16 : helloJShell()`

`/edit`	Invocazione editor esterno	`jshell> /edit helloJShell`

Come esempio più completo, proponiamo di seguito l'implementazione di un metodo che calcola il fattoriale di un numero ed è applicato agli elementi di un vettore per sostituirli con il valore calcolato.

```
$ jshell
|   Welcome to JShell -- Version 9-ea
|   For an introduction type: /help intro

jshell> int a[] = {0,1,3,5,8}
a ==> int[5] { 0, 1, 3, 5, 8 }

jshell> int fattoriale(int n){
  ...>    return n<2?1:n*fattoriale(n-1);
  ...> }
| created method fattoriale(int)

jshell> for (int i=0 ; i<a.length ; ++i)
  ...>    a[i] = fattoriale(a[i]);

jshell> a
a ==> int[5] { 1, 1, 6, 120, 40320 }

jshell> /exit
|  Goodbye
$
```

10 La documentazione automatica dei programmi con *Javadoc*

Uno degli aspetti critici dell'attività di programmazione è rappresentato dalla necessità di documentare adeguatamente il codice sviluppato. L'eccessiva personalizzazione delle tecniche di documentazione fa emergere la necessità di uno standard che stabilisca un insieme di regole condivise per annotare il codice sorgente, cosa tanto più importante quanto più l'applicazione software da sviluppare è complessa e articolata.

Per rispondere a questa esigenza l'ambiente di sviluppo standard di Java (JDK) comprende lo strumento a riga di comando *Javadoc*, che produce automaticamente una documentazione dei *package*, delle classi e dei relativi membri interpretando particolari commenti inseriti nel codice sorgente. La documentazione prodotta consiste in un insieme di pagine HTML navigabili mediante un qualsiasi *browser*; come dimostra la FIGURA **10**, la stessa documentazione della libreria standard del linguaggio disponibile in rete viene creata con questo strumento.

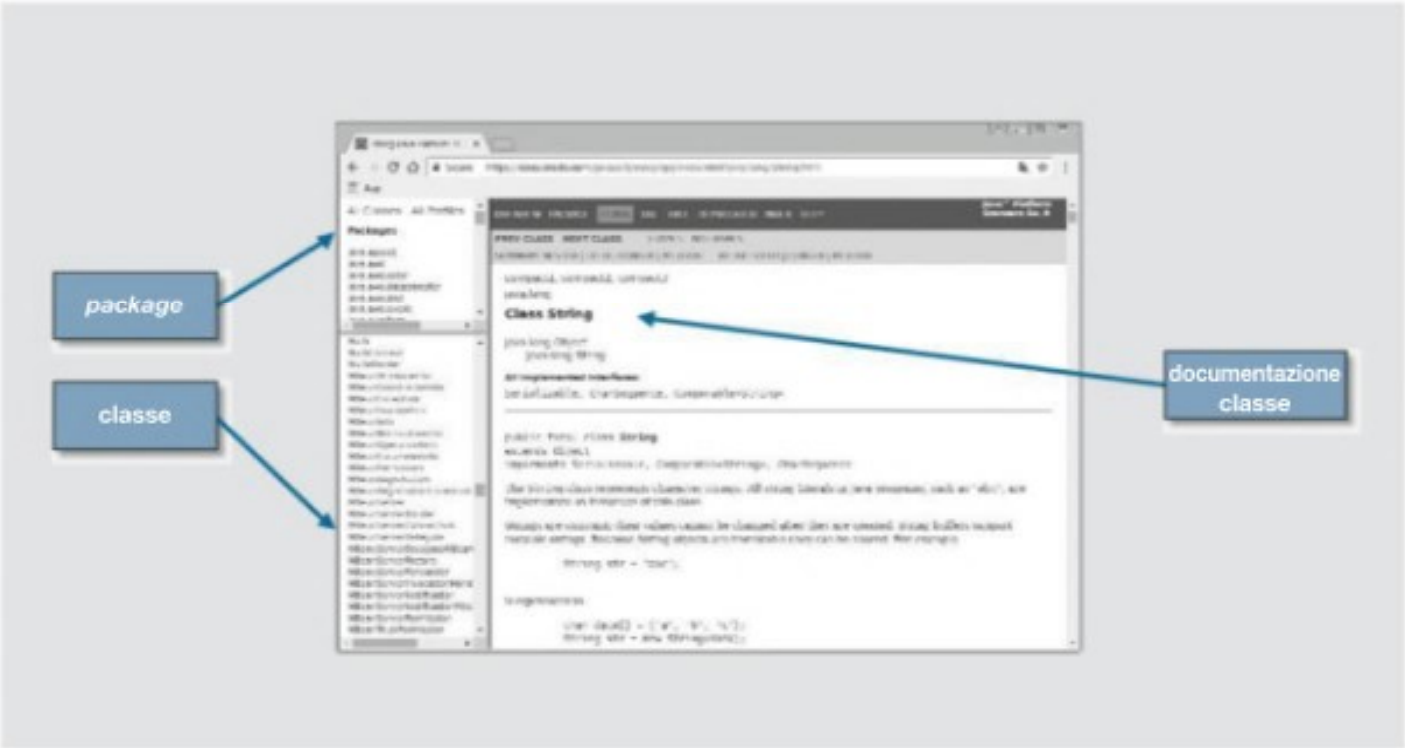


FIGURA **10**

Il programmatore che intende produrre questo tipo di documentazione per le classi che sviluppa deve inserire nel codice sorgente dei commenti «speciali» nel seguente formato:

```
/**
  * ...
  * ...
  * ...
*/
```

All'interno di questa tipologia di commenti, che sono interpretati dallo strumento *Javadoc*, lo sviluppatore deve inserire dei *tag*, ovvero parole chiave precedute dal simbolo «@» aventi lo scopo di qualificare il testo che le segue, che diverranno parte della documentazione HTML. La TABELLA 9 riporta i *tag Javadoc* fondamentali.

TABELLA 9

Tag	Significato
`@author`	Autore della classe.
`@version`	Versione della classe.
`@param parametro`	Descrizione del parametro di un metodo.
`@return`	Descrizione del valore di ritorno di un metodo.
`@exception` `@throws`	Descrizione di un'eccezione sollevata da un metodo.
`@see`	Inserzione di un riferimento.

ESEMPIO

La classe *Televisore* presentata all'inizio del capitolo può essere commentata per la generazione automatica della relativa documentazione con lo strumento *Javadoc*:

```
/**
  * classe dimostrativa per oggetti di tipo Televisore
  * @author F. F.
  * @version 1.0
  */
public class Televisore {
  private int pollici;
  private String schermo;
  private String colore;
  private int canale;
  private int volume;
  private int luminosita;
  private boolean acceso;
/**
  * crea un televisore con valori iniziali relativi a
  * numero di pollici, tipologia di schermo e colore esterno
  * gli attributi di stato sono impostati di default a:
  * canale=1, volume=10, luminosità=40, acceso=false
  * @param pollici numero di pollici
  * @param schermo tipologia di schermo (LCD, LED, …)
  * @param colore colore esterno del televisore (Nero, Bianco, …)
  */
public Televisore(int pollici, String schermo, String colore) {
  setPollici(pollici);
  setSchermo(schermo);
  setColore(colore);
  canale = 1;
  volume = 10;
  luminosita = 40;
  acceso = false;
}
/**
  * restituisce il numero di pollici del televisore
```

```
  * @return pollici
  */
public int getPollici() {return pollici;}
/**
  * imposta il numero di pollici del televisore
  * @param p dimensione in pollici
  */
private void setPollici(int p) {pollici = p;}
/**
  * restituisce il colore esterno del televisore
  */
public String getColore() {return colore;}
/**
  * imposta il colore esterno del televisore
  * @param c colore
  * @return colore
  */
private void setColore(String c) {colore = c;}
/**
  * restituisce la tipologia di schermo del televisore
  * @return schermo
  */
public String getSchermo() {return schermo;}
/**
  * imposta la tipologia di schermo del televisore
  * @param s tipo di schermo
  */
private void setSchermo(String s) {schermo = s;}
/**
  * restituisce il canale selezionato
  * @return canale
  */
public int getCanale() {return canale;}
/**
  * imposta direttamente un numero di canale
  * @param c valore numerico del canale (1-99)
  */
public void setCanale(int c) {if (c>0 && c<99) canale = c;}
/**
  * restituisce il livello di volume impostato
  * @return volume
  */
public int getVolume() {return volume;}
/**
  * restituisce il livello di luminosità impostato
  * @return luminosità
  */
public int getLuminosita() {return luminosita;}
/**
  * restituisce lo stato acceso/spento impostato
  */
public boolean isAcceso() {return acceso;}
/**
  * imposta a true l'attributo acceso
  */
```

```
public void accendi() {acceso = true;}
/**
  * imposta a false l'attributo acceso
  */
public void spegni() {acceso = false;}
/**
  * incrementa il numero di canale
  */
public void canaleSuccessivo() {
  if (canale<99) canale++;
}
/**
  * decrementa il numero di canale
  */
public void canalePrecedente() {
  if (canale>0) canale--;
}
/**
  * incrementa il livello di volume
  */
public void aumentaVolume() {if (volume<50) volume++;}
/**
  * decrementa il livello di volume
  */
public void diminuisciVolume() {
  if (volume>0) volume--;
}
/**
  * incrementa il livello di luminosità
  */
  public void aumentaLuminosita() {
    if (luminosita<80) luminosita++;
  }
  /**
    * decrementa il livello di luminosità
    */
  public void diminuisciLuminosita() {
    if (luminosita>0) luminosita--;
  }
}
```

Nelle FIGURE **11** e **12** è possibile osservare la documentazione HTML generata dallo strumento *Javadoc* a partire dal codice dell'esempio precedente.
Nella FIGURA **11**, in particolare, si possono vedere:

- nel frame di sinistra: l'elenco delle classi che compongono l'applicazione (in questo caso la sola classe *Televisore*);
- nel frame di destra:
  - in alto: le voci di menu che permettono di navigare sui singoli componenti dell'applicazione (barre colorate);
  - al centro: la descrizione della classe *Televisore* con i commenti, il sommario dei metodi costruttore e il sommario dei metodi previsti con relativa firma e commenti specifici. Ovviamente è possibile prendere visione di tutti i metodi utilizzando la barra di scorrimento verticale e quella di selezione orizzontale, visualizzare tutti i metodi, solo quelli concreti o solo quelli astratti (il significato di queste ultime due voci sarà chiarito nei prossimi capitoli).

In FIGURA **12**, a cui si accede effettuando lo *scroll* della pagina di FIGURA **10**, è invece possibile consultare i dettagli dei singoli metodi (in figura, costruttore e metodi *getPollici* e *getColore*).

FIGURA **11**

FIGURA **12**

Lo strumento *Javadoc* si esegue dalla riga di comando semplicemente elencando i file di codice sorgente di cui deve produrre la documentazione, in questo caso il solo file *Televisore.java*:

```
>javadoc Televisore.java
```

OSSERVAZIONE Le funzionalità dello strumento di documentazione automatica *Javadoc* sono presenti in molti IDE per il linguaggio Java, in modo da non dover ricorrere alla versione a riga di comando.

I CONCETTI CHIAVE

JAVA. Linguaggio di programmazione orientato agli oggetti e progettato per essere portabile tra piattaforme diverse.

BYTE-CODE. Il compilatore Java non produce direttamente codice eseguibile, ma un codice intermedio, *byte-code*, che viene eseguito dall'interprete Java.

JVM (*Java Virtual Machine*). È l'interprete che esegue il *byte-code* dei programmi Java. Ogni piattaforma hardware/software ha una sua propria specifica JVM per cui il *byte-code* è indipendente dalla piattaforma di sviluppo e di esecuzione, garantendo la portabilità («*write once, run anywhere*»). L'ambiente della JVM è strutturato in un set di istruzioni per il *byte-code*, un gruppo di registri per lo stato della macchina virtuale, un'area di memoria *stack* per l'esecuzione dei metodi, un'area di memoria *heap* per la creazione degli oggetti, e su cui opera il *garbage collector*, e un'area per la memorizzazione del codice dei metodi.

CLASSE. È l'elemento di base del linguaggio Java. Una classe è un insieme di dichiarazioni che costituiscono un modello formale per istanziare oggetti. La struttura generale di una classe prevede la dichiarazione del *package* di appartenenza e dei membri della classe (attributi e metodi).

PACKAGE. Meccanismo per organizzare le classi Java in gruppi ordinati. I *package* possono essere memorizzati in file compressi denominati file JAR.

OGGETTO. Istanza di una classe il cui valore degli attributi ne determina lo stato; agli oggetti specifici sono applicati i metodi della classe. A ogni oggetto viene assegnata al momento della creazione un'area nella memoria *heap* che viene recuperata dal *garbage collector* della JVM e resa nuovamente disponibile per essere riutilizzata quando l'oggetto risulta non più accessibile. Un oggetto viene istanziato con un'invocazione del tipo:

```
NomeClasse nomeOggetto =
  new NomeClasse(...);
```

ATTRIBUTI. Un attributo (o proprietà o variabile di istanza) è una variabile, o una costante, il cui valore in un determinato istante contribuisce a determinare lo stato di un oggetto.

METODI. I metodi sono funzioni applicate agli oggetti per realizzare l'aspetto operativo di una classe, permettendo l'accesso e la gestione dello stato interno determinato dagli attributi. L'invocazione di un metodo avviene con un'istruzione del tipo

```
nomeOggetto.nomeMetodo(...).
```

A differenza degli attributi che assumono valori distinti per i diversi oggetti istanza di una stessa classe, il codice dei metodi è comune a tutti gli oggetti di una classe e viene memorizzato una sola volta in una specifica area di memoria della JVM.

ACCESSIBILITÀ DEI MEMBRI DI UNA CLASSE. Java permette di definire diversi livelli di accessibilità dei membri (attributi e metodi) di una classe: privato, protetto, pubblico e *package*. Mediante i livelli di accessibilità viene implementata l'interfaccia degli oggetti rispettando il principio di *information hiding*.

MEMBRI STATICI DI UNA CLASSE. Java permette di definire i membri di una classe (attributi o metodi) con la qualifica `static`. Membri di questo tipo appartengono alla classe e non al singolo oggetto. Modificare il valore di un attributo statico significa modificare tale valore per tutte le istanze della classe. I metodi statici possono operare solo su attributi statici. Si può far riferimento ai membri statici di una classe con le seguenti notazioni:

```
NomeClasse.nomeAttributo
NomeClasse.nomeMetodo(...).
```

COSTRUTTORE. Particolare metodo di una classe che viene invocato all'atto della creazione di un oggetto invocando l'operatore `new`. Il costruttore ha sempre lo stesso nome della classe ed è sempre pubblico e non ha un tipo di ritorno in quanto «restituisce» un oggetto della classe stessa. La sua funzione è quella di inizializzare il valore degli attributi di un oggetto al momento della creazione. Se non viene implementato esplicitamente, Java crea un oggetto invocando un costruttore di default.

COSTRUTTORE DI COPIA. È un costruttore che accetta come argomento un oggetto della stessa classe: il suo scopo è quello di creare un nuovo oggetto il cui stato (valore degli attributi) è uguale a quello dell'oggetto passato come argomento (clonazione).

PROGRAMMA JAVA. È composto da un insieme di classi i cui oggetti cooperano tra loro scambiandosi messaggi attraverso i metodi.

METODO *MAIN*. È il metodo iniziale da cui parte l'esecuzione di un programma. L'intestazione del metodo *main* di una classe è sempre:

```
public static void main (String[] args) {...}
```

È bene che ogni classe disponga di un metodo *main* che verifichi la funzionalità dei metodi della classe stessa istanziandone un oggetto. In un programma formato da più classi l'esecuzione si avvia a partire dal metodo *main* della classe principale.

JAVABEAN. *Javabean* sono classi scritte in linguaggio Java che seguono particolari convenzioni. È previsto che esista un costruttore privo di argomenti e che l'accesso agli attributi privati risulti accessibile e modificabile tramite metodi il cui nome è dato dal nome dell'attributo stesso e dal prefisso *get* o *set*.

DATI PRIMITIVI E CLASSI *WRAPPER*. Nel linguaggio di programmazione Java, oltre ai tipi definiti dalle classi, esistono tipi primitivi per valori numerici (`int`, `long`, `short`, `byte`, `float`, `double`, ...), per i caratteri (`char`) e per i valori logici (`boolean`). In determinate situazioni può risultare utile trattare i dati primitivi come oggetti: a questo scopo Java rende disponibili specifiche classi ***wrapper*** nel *package java.lang*. Una classe *wrapper* comprende come attributo una variabile del tipo primitivo che incapsula, ovvero «trasforma» un tipo primitivo in un oggetto equivalente che dispone, attraverso i metodi della classe *wrapper* stessa, di nuove funzionalità.

CLASSI DI SERVIZIO. Sono classi che forniscono servizi di utilità senza la necessità di creare oggetti: non hanno costruttori e prevedono solo metodi statici. Un tipico esempio di classe di servizio Java è la classe *Math*, che rende disponibili numerosi metodi per il calcolo di funzioni matematiche (radici, logaritmi, esponenziali, funzioni trigonometriche, ...).

STRINGHE. In Java non esiste un tipo primitivo per le stringhe di caratteri, ma una classe predefinita denominata *String*. Un oggetto di tipo *String* è sempre costante: la modifica del suo valore implica la distruzione dell'oggetto stringa originale e la creazione di un nuovo oggetto stringa con il valore aggiornato. L'operatore «+» permette di concatenare tra loro due stringhe per crearne una nuova, il cui contenuto è dato dai caratteri della seconda giustapposti a quelli della prima. La classe *String* rende disponibili numerosi metodi per la gestione delle stringhe di caratteri.

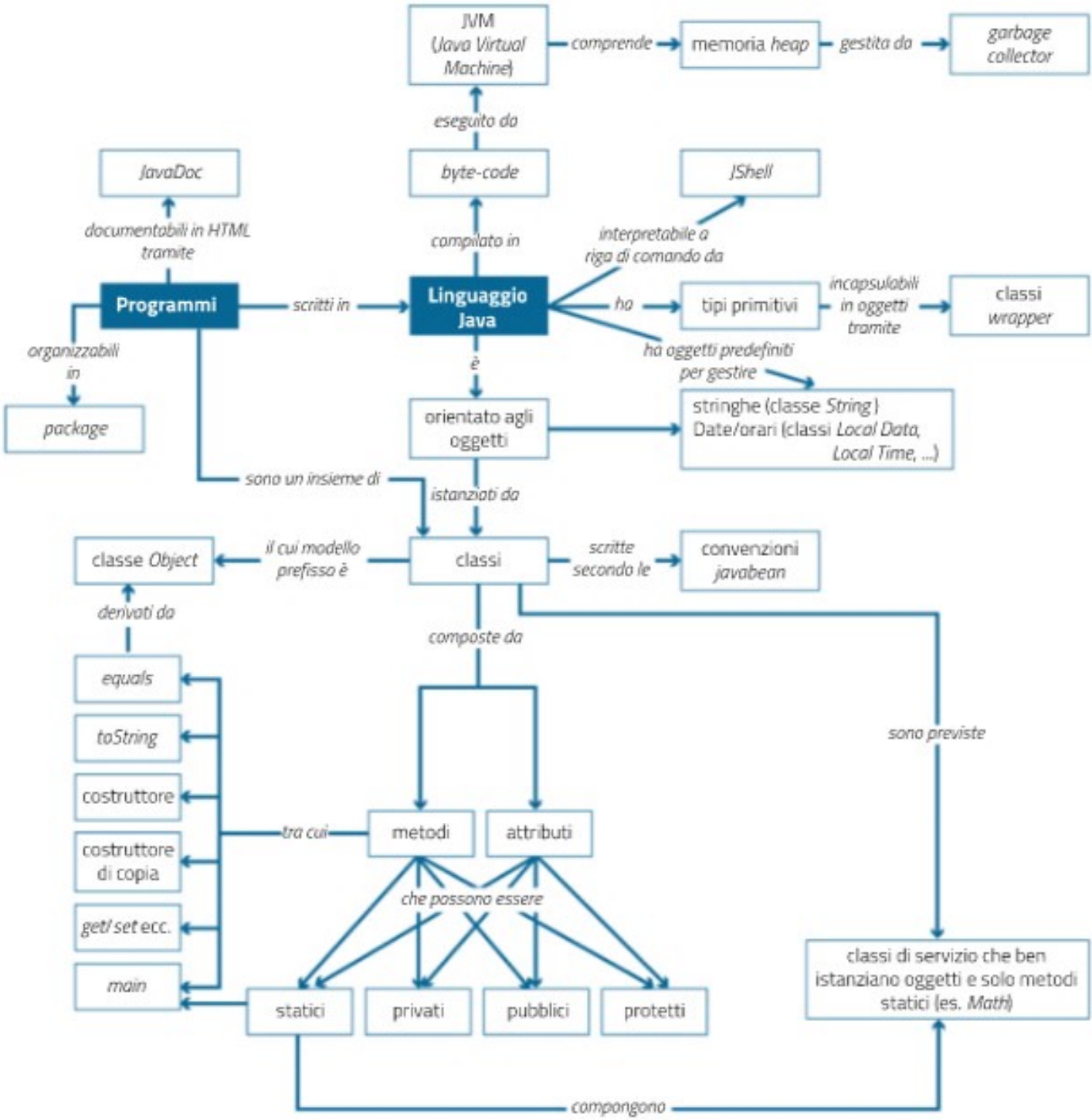
DATE E ORARI. Con la versione 8 di Java è stata rivista la gestione dei dati relativi a dati temporali (date e orari) con l'introduzione di una nuova API dedicata che offre classi: di oggetti immutabili rappresentanti un valore e non un intervallo temporale; *thread-safe* per una gestione automatica degli aspetti relativi alla concorrenza; con strumenti tali da poter usare date e orari in modo semplice e chiaro in contesti diversi; che prevedono la gestione di diversi sistemi di calendario per supportare le esigenze di particolari zone geografiche. I *package* previsti sono: *java.time* – contiene le classi base; *java.time.chrono* – gestisce differenti tipi di calendari; *java.time.format* – gestisce la formattazione e il *parsing* di date e orari; *java.time.temporal* – mette a disposizione classi per la manipolazione a basso livello di date e orari; *java.time.zone* – gestisce le *time zone* geografiche.

CODIFICA UNICODE. Set di caratteri che include, oltre all'intero standard ASCII, i caratteri utilizzati in molte lingue internazionali. Un carattere Unicode viene rappresentato in Java su due byte codificando oltre 65 000 simboli diversi.

JAVADOC. Strumento dell'ambiente di sviluppo del linguaggio Java che permette di automatizzare la generazione della documentazione dei programmi attraverso l'inserimento nel codice di commenti comprendenti specifici *tag* per documentazione degli elementi del codice (classi, metodi, parametri, ...). *Javadoc* produce un insieme di pagine HTML visualizzabili tramite un browser.

JAVA SHELL. *JShell*, introdotto con Java 9, è un ambiente interattivo per l'esecuzione a riga di comando di espressioni Java così da poter testare piccole parti di codice velocemente senza dover creare un progetto.

RIPASSA CON LA MAPPA



SPIEGA IN UNA PAGINA

1 Descrivere a che cosa si riferisce il motto «*write once, run anywhere*» e quali sono i meccanismi che stanno alla base di questa idea.

2 Descrivere sinteticamente cosa sono i membri statici di una classe e qual è il loro funzionamento.

QUESITI A RISPOSTA APERTA

3 Quali sono le caratteristiche principali della gestione dei dati di tipo stringa in Java?

4 Che cos'è l'ambiente *JShell*?

5 Qual è lo scopo del *garbage collector* di Java e come funziona?

6 Che cos'è l'ambiente *Javadoc* e quali sono le sue principali caratteristiche?

7 Che cosa prevede Java per la gestione di dati di tipo temporale?

8 A che cosa servono le classi *wrapper*?

9 Quali sono i livelli di accessibilità delle componenti di una classe Java?

10 Collegare i seguenti tipi predefiniti del linguaggio Java con la relativa implementazione:

`int`	*integer* 8 bit
`short`	*integer* 16 bit
`long`	*integer* 32 bit
`float`	*integer* 64 bit
`double`	*floating-point* 32 bit
`char`	*floating-point* 64 bit
`byte`	*Unicode*

TEST A RISPOSTA MULTIPLA

11 Quali delle seguenti è la forma generale per applicare un metodo a un oggetto?

A `nomeOggetto.nomeMetodo(...)`

B `nomeMetodo(...).nomeOggetto`

C `NomeClasse:nomeMetodo(...)`

D `nomeMetodo(nomeOggetto)`

12 Il *byte-code* è:

A il codice sorgente di un programma Java.

B il codice eseguibile generato dal compilatore Java.

C un codice intermedio generato dal compilatore Java che deve essere interpretato dalla JVM.

D un codice a basso livello di tipo *assembly*.

13 Quali delle seguenti affermazioni sono vere a proposito della *Java Virtual Machine* (JVM)?

A È indipendente dalla piattaforma hardware/software.

B Ogni tipo di piattaforma hardware/software ha una specifica JVM.

C La JVM è di fatto un interprete del *byte-code*.

D La JVM è un componente hardware.

14 La portabilità del codice Java è data dal fatto che:

A il *byte-code* sviluppato su una qualsiasi piattaforma hardware/software può essere eseguito su una qualsiasi piattaforma hardware/software.

B un programma sorgente Java può essere compilato con un qualsiasi compilatore, anche di un altro linguaggio di programmazione.

C la JVM sia sempre la stessa su qualsiasi piattaforma hardware/software.

D i programmi Java non sono compilati.

15 Il motto «*write once, run anywhere*» significa che il *byte-code* prodotto dalla compilazione Java può essere utilizzato su qualunque piattaforma:

A identica alla piattaforma di compilazione indipendentemente dalla JVM.

B hardware/software solo se fornita di JVM.

C hardware solo se ha lo stesso sistema operativo della piattaforma di compilazione e se fornita di JVM.

D software solo se ha lo stesso hardware della piattaforma di compilazione e se fornita di JVM.

16 **Una classe è:**

A una collezione di oggetti: collegando gli oggetti tra di loro si implementa l'incapsulamento.
B un modello da cui si creano oggetti simili.
C l'implementazione di un oggetto nella sintassi Java.
D un insieme di procedure dette metodi.

17 **Due oggetti diversi istanza della stessa classe condividono:**

A solo il valore degli attributi.
B solo i metodi.
C metodi e struttura generale.
D i metodi e il valore degli attributi.

18 **Quali delle seguenti affermazioni a proposito di un oggetto sono vere?**

A Le proprietà di un oggetto descrivono il suo stato.
B I metodi di un oggetto descrivono il suo stato.
C I metodi si riferiscono alle funzionalità di un oggetto.
D Un oggetto è un'astrazione di un oggetto reale.

19 **Attraverso quale meccanismo gli oggetti interagiscono tra di loro?**

A Lo scambio di messaggi attraverso l'invocazione dei metodi.
B Lo scambio di metodi attraverso l'inoltro di messaggi.
C La condivisione del valore degli attributi.
D La condivisione della funzionalità dei metodi.

20 **Qual è la funzione dell'operatore Java `new`?**

A Creare una classe.
B Creare un metodo.
C Creare un oggetto.
D Creare un *package*.

21 **È possibile creare due riferimenti distinti a uno stesso oggetto?**

A Sì, sempre.
B No, mai.
C Sì, se i riferimenti sono entrambi dello stesso tipo dell'oggetto che riferiscono.
D Sì, ma solo se i riferimenti non sono entrambi dello stesso tipo dell'oggetto che riferiscono.

22 **Nel linguaggio Java lo spazio di memoria assegnato agli attributi di un oggetto istanziato con l'operatore `new` viene liberato:**

A automaticamente dal supporto a tempo di esecuzione del linguaggio.
B solo al termine dell'esecuzione del programma.
C automaticamente dal distruttore della classe.
D automaticamente dal *garbage collector*.

23 **Quali delle seguenti affermazioni a proposito del metodo *main* di una classe sono vere?**

A È il primo metodo da cui si avvia l'esecuzione del codice di una classe.
B È il primo metodo che viene applicato ogni volta che si istanzia un oggetto.
C È un metodo statico.
D Non prevede parametri.

24 **Quali delle seguenti affermazioni a proposito di un costruttore sono vere?**

A È un metodo che viene eseguito automaticamente all'atto della creazione di un oggetto.
B Può prevedere come parametro un oggetto.
C Può non essere pubblico.
D Non prevede tipo di ritorno perché implicitamente restituisce un oggetto della classe.

25 **Il valore degli attributi dichiarati nella sezione privata di una classe:**

A può essere modificato mediante specifici metodi della classe stessa.
B può essere modificato solo dal costruttore.
C non può essere modificato.
D può essere modificato direttamente con un espressione del tipo *oggetto.attributo = espressione*.

26 **Quali delle seguenti affermazioni circa un attributo con accessibilità di livello `protected` sono vere?**

A La sua accessibilità è equivalente a `public`.
B La sua accessibilità diretta è possibile solo a livello di classe e classe derivata.
C La sua accessibilità diretta è possibile solo a livello di classe e di *package*.
D La sua accessibilità diretta è possibile solo a livello di classe, classe derivata e di *package*.

27 **I membri statici di una classe sono:**

A le costanti.

B gli attributi e i metodi che appartengono non alle singole istanze, ma alla classe.

C membri che possono essere riferiti con la notazione *NomeClasse.nomeMembro*.

D membri che possono essere riferiti con la notazione *nomeOggetto.nomeMembro*.

28 **Le classi *wrapper* dei tipi di dato primitivi sono:**

A gli stessi tipi di dato primitivi.

B classi che permettono di gestire valori associati ai tipi di dato primitivi come fossero oggetti.

C classi che permettono a oggetti di qualunque classe di gestire tipi di dato primitivi.

D classi che hanno come metodi gli operatori algebrici per definire espressioni con i tipi di dato primitivi.

29 **Indicare quanto vale in Java l'output generato dalla seguente istruzione**

```
System.out.println(3+4+" = "+3+4);
```

A `7 = 7`

B `7 = 34`

C `34 = 7`

D `34 = 34`

30 **Una stringa di caratteri in Java è:**

A un tipo di dato primitivo.

B un *array* di caratteri.

C un oggetto costante di classe *String*.

D un oggetto variabile di classe *String*.

31 ***Javadoc* è:**

A una classe i cui oggetti permettono di generare documenti.

B uno strumento Java per la documentazione automatica di programmi tramite *tag* inseriti nei commenti del codice da parte del programmatore.

C un insieme di specifiche a cui attenersi per scrivere programmi aderendo alle convenzioni di stile del linguaggio Java.

D il documento di rilascio delle varie versioni di Java.

32 ***Javabean* è:**

A una classe i cui oggetti permettono di sviluppare documenti.

B uno strumento Java per la documentazione automatica dei programmi.

C una classe Java che osserva alcune convenzioni di codifica.

D la classe da cui derivano tutte le classi del linguaggio Java.

PROBLEMI

33 **Si intende definire una classe *Persona* che abbia come attributi *età, nome, sesso, professione*. Realizzare il diagramma UML della classe e la relativa implementazione in linguaggio Java prevedendo oltre ai metodi *getter* e *setter* per ogni attributo:**

- **un costruttore che permetta di istanziare oggetti di tipo *Persona* definendo valori specifici per i vari attributi;**
- **il metodo *chiSei* che restituisca, quando invocato, una stringa del tipo «Sono una persona di nome: *nome*, sesso: *sesso*, età: *età*, professione: *professione*» dove *nome, sesso, età* e *professione* sono i valori assunti dagli attributi della classe;**
- **il metodo *main* che preveda la creazione di un oggetto di tipo *Persona* e l'invocazione del metodo *chiSei* per visualizzarne il risultato restituito.**

34 **Si devono gestire degli oggetti di tipo *Angolo* nella forma g°p's" dove *g* rappresenta i gradi, *p* i primi ed *s* i secondi (con $0 \le g < 360$, $0 \le p < 60$, $0 \le s < 60$). Dopo aver prodotto il diagramma UML della classe *Angolo* la si implementi in linguaggio Java rendendo disponibili i seguenti metodi (*n* rappresenta un valore intero, mentre *a* rappresenta un oggetto di tipo *Angolo*):**

`visualizzaAngolo()`	visualizza l'angolo nel formato g°p's".
`aggiungiGradi(n)`	aggiunge all'angolo *n* gradi
`aggiungiPrimi(n)`	aggiunge all'angolo *n* primi
`aggiungiSecondi(n)`	aggiunge all'angolo *n* secondi
`angoloSecondi()`	ritorna il valore dell'angolo espresso in secondi
`secondiAngolo(n)`	imposta il valore dell'angolo (gradi, primi e secondi) corrispondente a *n* secondi
`differenzaSecondi(a)`	restituisce la differenza espressa in secondi tra l'angolo e l'angolo *a*
`sommaAngolo(a)`	somma all'angolo l'angolo *a*

Si definisca inoltre un costruttore che accetti in ingresso tre parametri interi per il valore dei gradi, dei primi e dei secondi e un costruttore di copia. Quando aggiornati i valori degli attributi devono essere normalizzati rispettando i limiti imposti (360 per i gradi, 60 per i primi e i secondi). Implementare infine il metodo *main* dove sono istanziati due o più oggetti di tipo *Angolo*, in modo da verificare l'invocazione di ogni singolo metodo in diverse condizioni.

35 Progettare mediante un diagramma UML e implementare in linguaggio Java una classe i cui oggetti rappresentano programmi per computer. Ogni oggetto deve avere almeno i seguenti attributi: *denominazione, produttore, versione, sistema operativo, anno*. La classe deve avere almeno i seguenti metodi:

- costruttore che ha come parametri *denominazione, produttore, versione, sistema operativo, anno*;
- costruttore di copia;
- *getDenominazione, getProduttore, getVersione, getSistema* e *getAnno* che restituiscono i valori degli attributi relativi;
- *setDenominazione, setProduttore, setVersione, setSistema* e *setAnno* che modificano i valori degli attributi relativi;
- *toString* che restituisce una stringa con tutti i dati dell'oggetto su cui è invocato;
- *compareAnno* che consente di confrontare l'anno di rilascio del programma con l'anno di rilascio di un altro programma.

Inserire nella classe un metodo *main* che consenta di verificarne tutte le funzionalità.

36 Progettare mediante un diagramma UML e implementare in linguaggio Java una classe *CD* i cui oggetti rappresentano CD audio. Ogni oggetto *CD* deve avere almeno le seguenti caratteristiche: *titolo, autore, numero brani, durata*. La classe deve avere i seguenti metodi:

- costruttore che ha come parametri *titolo, autore, numero brani* e *durata*;
- *getTitolo, getAutore, getDurata* e *getBrani* che restituiscono i valori degli attributi relativi;
- *setTitolo, setAutore, setDurata* e *setBrani* che modificano i valori degli attributi relativi;
- *toString* che restituisce una stringa con tutti i dati dell'oggetto su cui è invocato;
- *compareDurata* che consente di confrontare la durata complessiva del CD con la durata complessiva di un altro CD.

Inserire nella classe un metodo *main* che consenta di verificarne tutte le funzionalità.

37 A SCUOLA DI LAVORO Una catena di autonoleggio deve gestire con un sistema informatico i propri veicoli; per ogni veicolo devono essere memorizzate le seguenti informazioni:

- targa;
- marca;
- modello;
- cilindrata;
- anno di acquisto;
- numero di posti.

Definire mediante un diagramma UML e implementare in linguaggio Java una classe per rappresentare gli oggetti di tipo *Veicolo* che rispetti le specifiche *Javabean* prevedendo un metodo *main* per testarne le funzionalità.

38 A SCUOLA DI LAVORO Un'organizzazione deve catalogare i computer utilizzati dai propri dipendenti; per ogni computer devono essere memorizzate le seguenti informazioni:

- codice;
- marca;
- modello;
- velocità del processore;
- dimensioni della memoria RAM;
- dimensioni del disco;
- dimensioni del monitor;
- anno di acquisto.

Il codice di ogni computer è un numero progressivo generato automaticamente e non ulteriormente modificabile. Definire mediante un diagramma UML e implementare in linguaggio Java una classe per rappresentare gli oggetti di tipo *Computer* che rispetti le specifiche *Javabean* prevedendo un metodo *main* per testarne le funzionalità.

CASI PRATICI

39 Progettare mediante un diagramma UML e implementare, mediante una classe Java che rispetti le specifiche *Javabean*, una classe per la rappresentazione di un punto sulla superficie della Terra individuato dalle proprie coordinate geografiche (latitudine e longitudine espresse in gradi, primi e secondi).

40 Progettare mediante un diagramma UML e implementare in linguaggio Java una classe per la gestione di rettangoli da visualizzare su un display LCD monocromatico di 800 × 600 pixel (i rettangoli hanno i lati paralleli ai bordi del display e di essi sono visualizzati solo i lati) ipotizzando di disporre della seguente classe *Segmento* di FIGURA **13**.

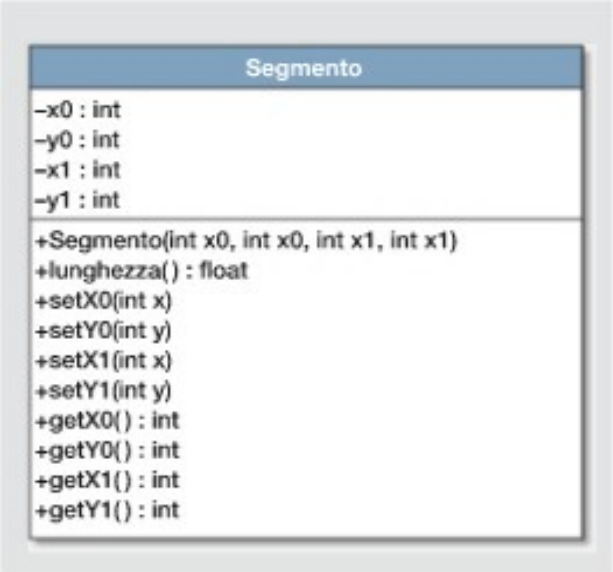


FIGURA **13**

La classe *Rettangolo* deve esporre – oltre ai metodi *setter* e *getter* – metodi per:

- calcolare l'area in pixel del rettangolo;
- determinare se il rettangolo interseca o meno un secondo rettangolo;
- spostare il rettangolo lungo le direzioni parallele ai lati del display;
- ruotare il rettangolo di 90° rispetto al proprio centro.

La classe deve inoltre disporre di un metodo *main* che consenta un test adeguato delle proprie funzionalità.

41 Modificare le classi *Segmento* e *Rettangolo* dell'esercizio precedente in modo da adattarle per l'uso con un display LCD a colori; ipotizzare attributi e metodi della classe *Colore*.

42 Un vettore nel piano cartesiano è un segmento orientato definito dalle coordinate dei due estremi («origine» e «vertice») e dal loro ordinamento: due vettori con estremi coincidenti diversamente orientati sono sovrapposti, ma diversi! Implementare in linguaggio Java la classe di FIGURA **14**, definita mediante un diagramma UML e relativa a un vettore nel piano cartesiano. Inserire nella classe un metodo *main* che consenta di verificarne tutte le funzionalità.

FIGURA **14**

lexicographic
quello lessicografico è un criterio di ordinamento di stringhe equivalente a quello utilizzato nei dizionari e che può essere esteso a un qualsiasi insieme di simboli

Mary Poppins
personaggio dell'omonimo film del 1964 diretto da Robert Stevenson e basato sui romanzi scritti da Pamela Lyndon Travers

passby-value
letteralmente «passare accanto a». Per una variabile stringa è rappresentato non tanto dal suo valore effettivo ma solo dal riferimento per giungere a esso

span
distanza, intervallo

5.6 The standardised String objects

5.6.1 Declaring and assigning String variables

In Java, a string of characters is manipulated with an object of type `String`. The String type does *not* belong to the primitive types (`boolean`, `int`, `double`, etc.), but is rather a non-primitive object type. Therefore, a variable of type String is a reference to that object in the global memory. Thus for the same reasons as for the case of integers, the following `badSwap` function will not swap the string variables since once the function is executed the values of p and q (that is, the respective references) are kept unchanged: Java is passby-value/reference only. For non-primitive types such as arrays, objects and strings, the stored value is a reference.

```
static void badSwap(String p, String q)
{  String tmp;
   tmp=p;
   p=q;
   q=tmp;}
```

To declare and initialise a `String`2, we proceed as follows

```
String title="First Hands-on Programming!";
String anotherTitle=title;
...
title="Extreme programming sounds better!"; // String can be
                                            // modified
To print a string, use the regular System.out.println() function:
System.out.println(title);
```

Java proposes many standard functions for efficiently manipulating strings. Let us next review only the very fundamental functions that operate on `String` objects.

5.6.2 Length of a string: length()

The length of a string object is defined as its number of characters. The length of a string is reported by calling the `length()` method on the `String` object, as shown below:

```
String s="Supercalifragilisticexpialidocious";
System.out.println(s.length());
```

We get 34, which is the longest invented word in a song from Mary Poppins written by the Sherman Brothers in 1964. Observe that the syntax is different from arrays. For an array, we get its length (that is, its number of elements) by the following syntax: `array.length`. But for a string, we use the method `length()` (with no argument, which explains the () parentheses).

5.6.3 Equality test for strings: equals(String str)

To test whether two strings s1 and s2 are identical or not, do not use the regular == equality test for primitive types. Indeed, testing whether two strings are identical or not by using the comparison == will check whether the references of these strings are identical or not. Instead, to test whether the contents of the two strings are identical or not, use the method `equals(str)` with a string argument str. Considers for example, the following sequence of instructions:

```
String s1="java";
String s2="JAVA";
System.out.println(s1.equals(s2));
```

lexicographic
quello lessicografico è un criterio di ordinamento di stringhe equivalente a quello utilizzato nei dizionari e che può essere esteso a un qualsiasi insieme di simboli

Mary Poppins
personaggio dell'omonimo film del 1964 diretto da Robert Stevenson e basato sui romanzi scritti da Pamela Lyndon Travers

passby-value
letteralmente «passare accanto a». Per una variabile stringa è rappresentato non tanto dal suo valore effettivo ma solo dal riferimento per giungere a esso

span
distanza, intervallo

5.6 The standardised String objects

5.6.1 Declaring and assigning String variables

In Java, a string of characters is manipulated with an object of type `String`. The String type does *not* belong to the primitive types (`boolean`, `int`, `double`, etc.), but is rather a non-primitive object type. Therefore, a variable of type String is a reference to that object in the global memory. Thus for the same reasons as for the case of integers, the following `badSwap` function will not swap the string variables since once the function is executed the values of p and q (that is, the respective references) are kept unchanged: Java is passby-value/reference only. For non-primitive types such as arrays, objects and strings, the stored value is a reference.

```
static void badSwap(String p, String q)
{  String tmp;
   tmp=p;
   p=q;
   q=tmp;}
```

To declare and initialise a `String`2, we proceed as follows

```
String title="First Hands-on Programming!";
String anotherTitle=title;
...
title="Extreme programming sounds better!"; // String can be
                                            // modified
To print a string, use the regular System.out.println() function:
System.out.println(title);
```

Java proposes many standard functions for efficiently manipulating strings. Let us next review only the very fundamental functions that operate on `String` objects.

5.6.2 Length of a string: length()

The length of a string object is defined as its number of characters. The length of a string is reported by calling the `length()` method on the `String` object, as shown below:

```
String s="Supercalifragilisticexpialidocious";
System.out.println(s.length());
```

We get 34, which is the longest invented word in a song from Mary Poppins written by the Sherman Brothers in 1964. Observe that the syntax is different from arrays. For an array, we get its length (that is, its number of elements) by the following syntax: `array.length`. But for a string, we use the method `length()` (with no argument, which explains the () parentheses).

5.6.3 Equality test for strings: equals(String str)

To test whether two strings s1 and s2 are identical or not, do not use the regular == equality test for primitive types. Indeed, testing whether two strings are identical or not by using the comparison == will check whether the references of these strings are identical or not. Instead, to test whether the contents of the two strings are identical or not, use the method `equals(str)` with a string argument str. Considers for example, the following sequence of instructions:

```
String s1="java";
String s2="JAVA";
System.out.println(s1.equals(s2));
```

We get `false` on the console output since Java is *case-sensitive*. That is, upper cases are considered different from lower cases in Java. We can access the character of the string at position $k-1$ by calling the method `charAt(k)`. Again, indexes of characters range from 0 to the string length minus one. For example, consider the following code:

```
String s="3.14159265";
System.out.println(s.charAt(1));
```

Then the executed program writes the separation "." on the console. We can assign a character variable with that string character as follows:

```
char c=s.charAt(1);
```

5.6.4 Comparing strings: Lexicographic order

Before comparing strings of various lengths, we first need to define what is meant by the comparison of two elementary characters. The "distance" between two characters is defined as the span in the ASCII3 code table. Since Java is case-sensitive, lower cases are different from upper cases (they have different integer codes), and the same character appears twice in the ASCII table: once for the upper case and once for the lower case.

Program 5.8 Lower/upper cases and ASCII codes of characters

```
char c1, c2;
c1='a';
c2='z';
// Compare character code
if ( c1<c2 )
{ System.out.println( c1+" is before "+c2 ); } else
{ System.out.println( c1+" is after or equal to "+c2 ); }
int codec1=c1; // type casting conversion
int codec2=c2; // type casting conversion
System.out.println( "Code ASCII for "+c1+":"+codec1 );
System.out.println( "Code ASCII for "+c2+":"+codec2 );
```

Running this code, we get

```
a is before z
Code ASCII for a:97
Code ASCII for z:122
```

[F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Springer-Verlag London Limited, 2009]

QUESTIONS

a **Is the Java `String` type a primitive or a non-primitive object type?**

b **Which method is used to calculate the length of a string?**

c **What is the correct way to compare two `String` objects?**

d **How is the «distance» between two characters defined?**

A3

La programmazione orientata agli oggetti in Java

Nel precedente capitolo è stata introdotta come esempio la classe *Libro*; supponiamo di dover modellare una mensola sopra la quale collocare dei libri. Un possibile diagramma UML potrebbe essere quello di FIGURA 1.

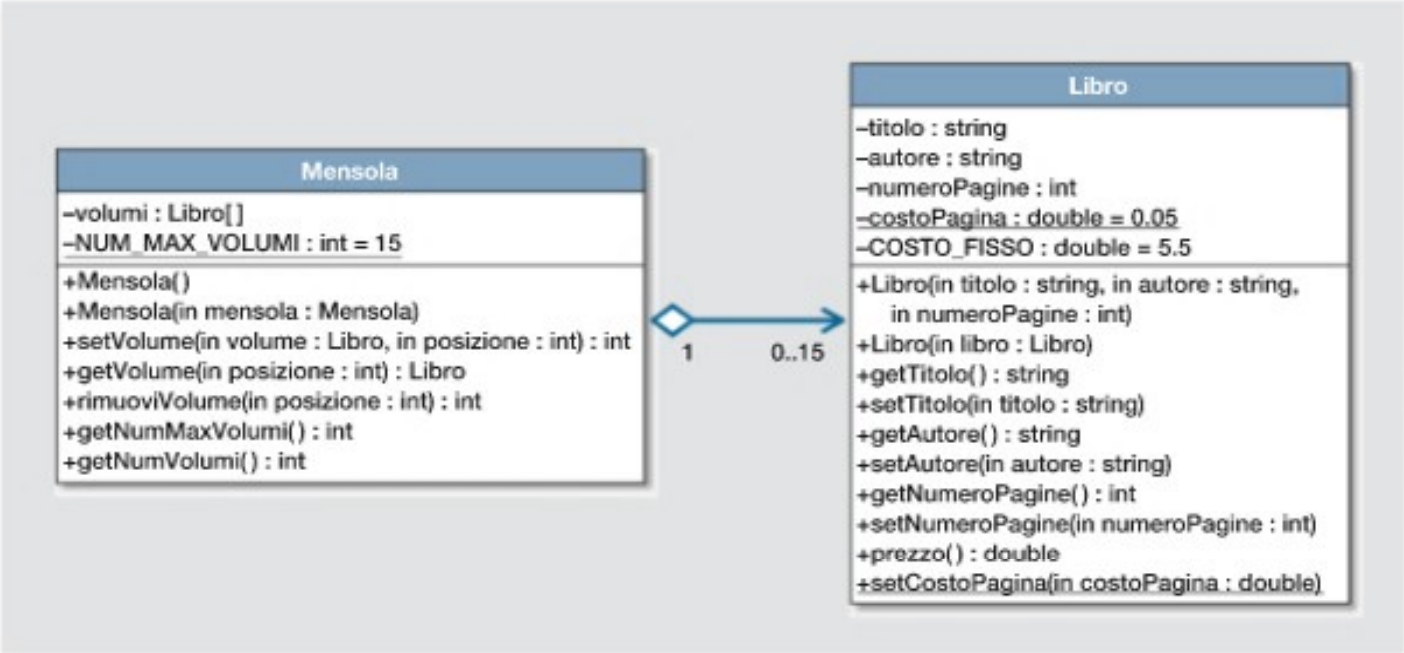


FIGURA 1

La mensola è in questo caso un «contenitore» di volumi disposti uno di seguito all'altro: ogni mensola prevede un numero fisso di posizioni per i volumi – nell'esempio è 15 – ognuna delle quali può essere vuota o contenere un libro. L'associazione tra le due classi modella un'aggregazione perché un oggetto di classe *Mensola* può esistere indipendentemente dal fatto che contenga o meno dei libri.

Riprendendo l'implementazione Java della classe *Libro*:

```
public class Libro {
  // attributi
  private String titolo;
  private String autore;
  private int numeroPagine;
  private static double costoPagina=0.05;
  final double COSTO_FISSO=5.5;

  // costruttori
  public Libro(String titolo, String autore, int numeroPagine) {
    this.titolo=titolo;
```

```
    this.autore=autore;
    this.numeroPagine=numeroPagine;
  }

  public Libro(Libro libro) {
    this.titolo=libro.getTitolo();
    this.autore=libro.getAutore();
    this.numeroPagine=libro.getNumeroPagine();
  }

  // metodi
  public void setTitolo(String titolo) {this.titolo=titolo;}
  public String getTitolo() {return titolo;}

  public void setAutore(String autore) {this.autore=autore;}
  public String getAutore() {return autore;}

  public void setNumeroPagine(int pagine) {numeroPagine=pagine;}
  public int getNumeroPagine() {return numeroPagine;}

  public double prezzo() {
    return COSTO_FISSO+numeroPagine*costoPagina;
  }
  public static void setCostoPagina(double costo) {
    costoPagina=costo;
  }
}
```

Avremo in Java la seguente classe *Mensola*:

```
public class Mensola {
  // attributi
  private static final int MAX_NUM_VOLUMI=15;
  private Libro volumi[];

  // costruttori
  public Mensola() {
    volumi=new Libro[MAX_NUM_VOLUMI];
  }

  public Mensola(Mensola mensola) {
    int posizione;

    volumi=new Libro[MAX_NUM_VOLUMI];
    for(posizione=0; posizione<MAX_NUM_VOLUMI; posizione++) {
      if (mensola.getVolume(posizione)!= null)
        volumi[posizione] = mensola.getVolume(posizione);
    }
  }

  // metodi
  public int setVolume(Libro libro, int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return -1; // posizione non valida
    if (volumi[posizione]!= null)
      return -2; // posizione occupata
    volumi[posizione]=libro; // inserimento libro in posizione
```

```
    this.autore=autore;
    this.numeroPagine=numeroPagine;
  }

  public Libro(Libro libro) {
    this.titolo=libro.getTitolo();
    this.autore=libro.getAutore();
    this.numeroPagine=libro.getNumeroPagine();
  }

  // metodi
  public void setTitolo(String titolo) {this.titolo=titolo;}
  public String getTitolo() {return titolo;}

  public void setAutore(String autore) {this.autore=autore;}
  public String getAutore() {return autore;}

  public void setNumeroPagine(int pagine) {numeroPagine=pagine;}
  public int getNumeroPagine() {return numeroPagine;}

  public double prezzo() {
    return COSTO_FISSO+numeroPagine*costoPagina;
  }
  public static void setCostoPagina(double costo) {
    costoPagina=costo;
  }
}
```

Avremo in Java la seguente classe *Mensola*:

```
public class Mensola {
  // attributi
  private static final int MAX_NUM_VOLUMI=15;
  private Libro volumi[];

  // costruttori
  public Mensola() {
    volumi=new Libro[MAX_NUM_VOLUMI];
  }

  public Mensola(Mensola mensola) {
    int posizione;

    volumi=new Libro[MAX_NUM_VOLUMI];
    for(posizione=0; posizione<MAX_NUM_VOLUMI; posizione++) {
      if (mensola.getVolume(posizione)!= null)
        volumi[posizione] = mensola.getVolume(posizione);
    }
  }

  // metodi
  public int setVolume(Libro libro, int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return -1; // posizione non valida
    if (volumi[posizione]!= null)
      return -2; // posizione occupata
    volumi[posizione]=libro; // inserimento libro in posizione
```

```
    return posizione; // restituisce la posizione di inserimento
  }

  public Libro getVolume(int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return null; // posizione non valida: nessun libro
                   // restituito
    return volumi[posizione]; // restituisce il libro
                              // in posizione
  }

  public int rimuoviVolume(int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return -1; // posizione non valida
    if (volumi[posizione]== null)
      return -2; // posizione vuota
    volumi[posizione]= null; // rimozione libro in posizione
    return posizione; // restituisce la posizione liberata
  }

  public int getNumMaxVolumi() {
    return MAX_NUM_VOLUMI;
  }

  public int getNumVolumi() {
    int posizione, n=0;
    for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
      if (volumi[posizione]!= null)
        n++;
    return n;
  }
}
```

Nell'esempio appena visto si può notare che:

- Una *mensola* incapsula un *array* di oggetti Java: ogni singola posizione dell'*array volumi* può contenere un oggetto di tipo *Libro*, o essere vuota (`null`).
- Esistono due costruttori: il primo istanzia un oggetto di classe *Mensola* privo di libri, mentre il secondo istanzia un oggetto di tipo *Mensola* contenente gli stessi libri di un oggetto della stessa classe *Mensola* preesistente e fornito come parametro (costruttore di copia).
- Il metodo *setVolume* inserisce un volume in una posizione della mensola specificata mediante un indice se questa è valida (indice compreso tra 0 e il numero massimo di volumi previsto per la mensola) oppure vuota (`null`). Se la posizione espressa dall'indice non è valida il metodo restituisce il valore negativo –1; se la posizione espressa dall'indice è valida ma non è vuota restituisce il valore negativo –2; se l'operazione è eseguita con successo restituisce l'indice della posizione (un valore sempre positivo o uguale a 0).
- Il metodo *getVolume* restituisce un oggetto di tipo *Libro* da una determinata posizione della mensola specificata mediante un indice se

questa è valida (indice compreso tra 0 e il numero massimo di volumi previsto per la mensola); in caso contrario – o se la posizione è vuota (`null`) – il metodo restituisce il valore `null`.

- Il metodo *rimuoviVolume* libera una determinata posizione della mensola specificata mediante un indice se questa è valida (secondo gli stessi criteri dei metodi *setVolume* e *getVolume*) e non vuota (`null`); se la posizione non è valida il metodo restituisce il valore negativo –1; se la posizione è valida, ma vuota, restituisce il valore negativo –2; se l'operazione è eseguita con successo (viene posto a `null` il valore contenuto nella posizione indicata) restituisce l'indice della posizione resa disponibile.
- I metodi *getNumMaxVolumi* e *getNumVolumi* restituiscono rispettivamente il numero massimo di volumi che la mensola può contenere (nel codice si fa sempre riferimento al valore dell'attributo costante *NUM_MAX_VOLUMI* in modo che, volendo adattare la classe a rappresentare mensole di dimensione diversa, è sufficiente modificare solo l'inizializzazione di questo attributo) e il numero dei volumi effettivamente presenti su una mensola nel momento dell'invocazione.

1 Gli *array* in Java

Gli *array* sono dichiarati nel linguaggio di programmazione Java con la seguente sintassi:

```
tipo[] identificatore;
```

Questa dichiarazione definisce un vettore di elementi di tipo *tipo* (un tipo primitivo o una classe); la coppia di simboli giustapposti «[» e «]» che identifica il vettore può essere posta anche dopo il nome dell'identificatore:

```
tipo identificatore[];
```

ESEMPIO Le seguenti dichiarazioni definiscono entrambe un vettore di valori interi (tipo primitivo `int`) denominato *unVettore*:

```
int[] unVettore;
int unVettore[];
```

Diversamente da altri linguaggi – come per esempio C++ – la **dichiarazione** di un *array* in Java comporta la sola allocazione della variabile che contiene il riferimento a un *array*, ma non l'allocazione dell'*array* vero e proprio. Per questo motivo essa non prevede la specifica delle dimensioni, cioè del numero di elementi, dell'*array*. Nel caso dell'esempio precedente, *unVettore* è solo una variabile dichiarata in modo tale da poter contenere il riferimento a un *array*.

Dal momento che in Java gli *array* sono oggetti, l'*array* vero e proprio viene creato – normalmente nel costruttore di una classe – utilizzando l'operatore `new`.

ESEMPIO

L'istruzione

```
unVettore = new int[10];
```

a seguito della dichiarazione dell'esempio precedente crea un *array* di 10 elementi di tipo intero all'interno dell'area di memoria *heap* che nella JVM è riservata all'allocazione degli oggetti.

L'esempio precedente è relativo a un *array* di elementi di un tipo di dato primitivo (specificatamente `int`) in cui i singoli elementi dell'*array* contengono direttamente i valori.

Nel caso di dichiarazione di *array* di oggetti di una determinata classe, i singoli elementi dell'*array* contengono i **riferimenti** agli oggetti.

ESEMPIO

Nell'area di memoria *heap* della JVM, dopo l'esecuzione dell'istruzione

```
Libro[] volumi=new Libro[10];
```

si incontrerà una situazione come quella che segue:

Supponendo di eseguire le seguenti istruzioni

```
Libro l1=new Libro("Pinocchio", "C. Collodi", 150);
Libro l2=new Libro("Pollicino", "C. Perrault", 80);
volumi[0]=l1;
volumi[2]=l2;
```

la situazione diviene quella di FIGURA **2**.

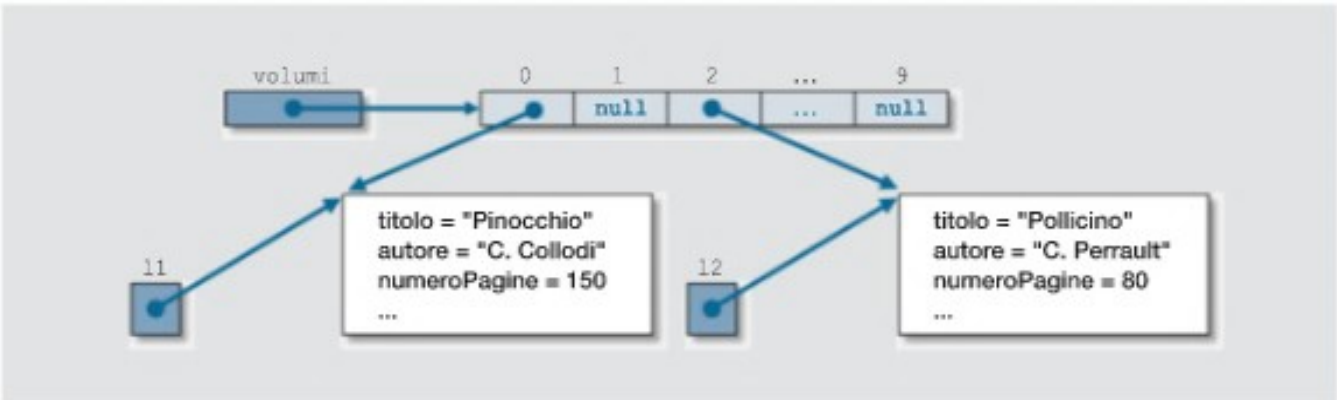


FIGURA **2**

Se successivamente viene eseguita la seguente istruzione

```
volumi[0] = null;
```

si avrà la situazione di FIGURA 3.

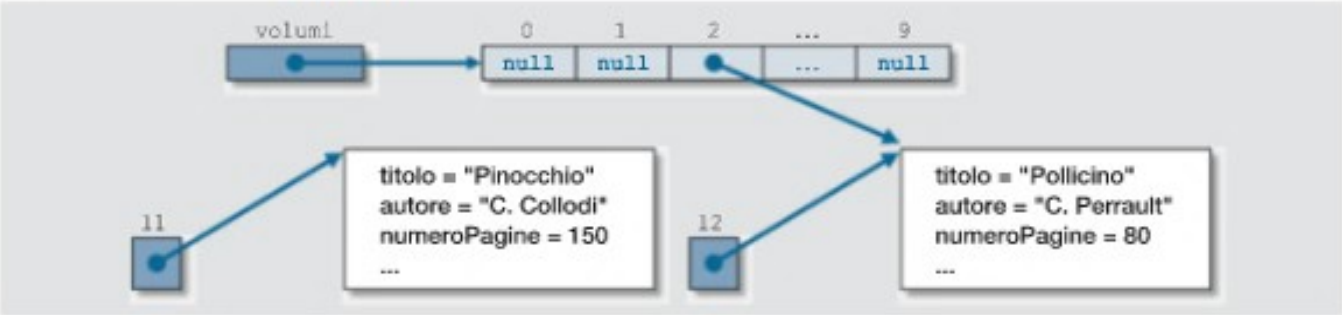


FIGURA 3

Se invece l'inizializzazione dei vettori fosse stata effettuata mediante le seguenti istruzioni

```
volumi[0] = new Libro("Pinocchio", "C. Collodi", 150);
volumi[2] = new Libro("Pollicino", "C. Perrault", 80);
```

la situazione nello *heap* sarebbe stata quella di FIGURA 4.

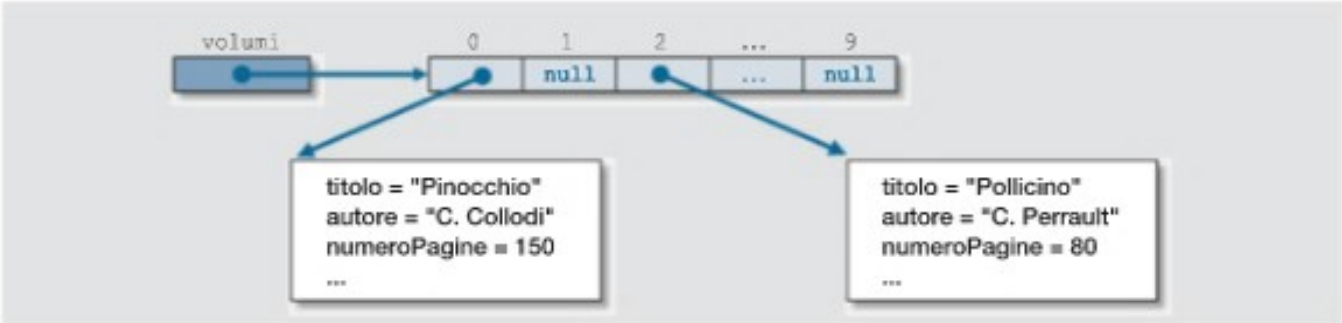


FIGURA 4

Quindi, dopo l'esecuzione dell'istruzione

```
volumi[0] = null;
```

sarebbe stata quella di FIGURA 5, dove l'oggetto di classe *Libro* a cui veniva fatto riferimento dall'elemento di indice 0, non essendo più riferito da alcuna variabile, è stato rimosso dal *garbage collector* della JVM.

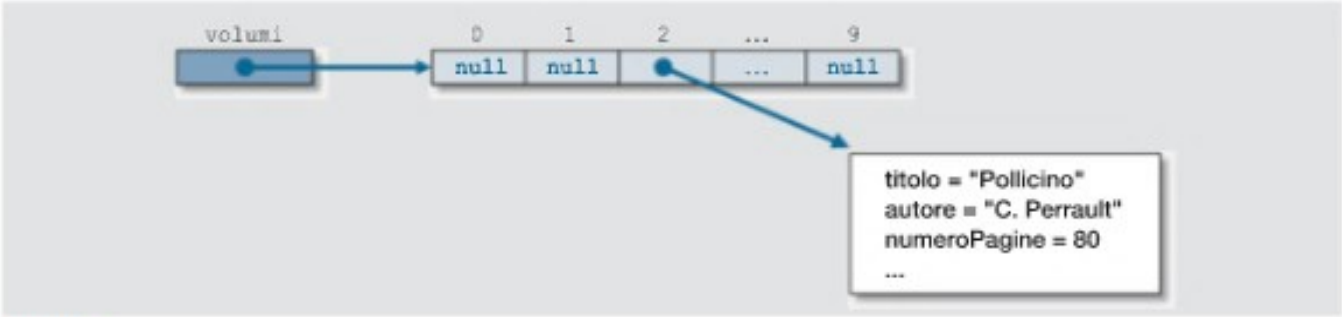


FIGURA 5

Dovendo utilizzare un *array* a più dimensioni la sintassi è la seguente:

```
tipo[][]...[] identificatore;
```

ESEMPIO

Per definire una matrice di valori interi è possibile usare la seguente dichiarazione

```
int[][] tavolaPitagorica;
```

che dovrà essere seguita dalla seguente istruzione per istanziare l'oggetto matrice

```
tavolaPitagorica = new int [10][10];
```

Per quanto riguarda le modalità di riferimento dei singoli elementi di un *array*, sia esso monodimensionale o multidimensionale, esse adottano la stessa sintassi di altri linguaggi di programmazione, come per esempio C/C++, utilizzando i simboli «[» e «]» per specificare l'indice.

ESEMPIO Il seguente metodo *cercaTitolo* aggiunto alla classe *Mensola* consente di determinare se su una mensola è presente o meno un volume avente il titolo specificato:

```
public boolean cercaTitolo (String titolo) {
  int posizione;

  for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
    if (volumi[posizione]!= null)
      if (volumi[posizione].getTitolo().equals(titolo))
        return true;
  return false;
}
```

OSSERVAZIONE Dato che un *array* può essere un *array* di oggetti e che in Java un *array* è esso stesso un oggetto, è possibile creare strutture dati a vari livelli di complessità.

ESEMPIO A partire dalle classi *Libro* e *Mensola* possiamo pensare di sviluppare una classe *Scaffale* strutturata come un insieme ordinato di 5 mensole; in sintesi uno scaffale è un oggetto composto da 5 mensole ciascuna delle quali può contenere fino a 15 libri (FIGURA 6).

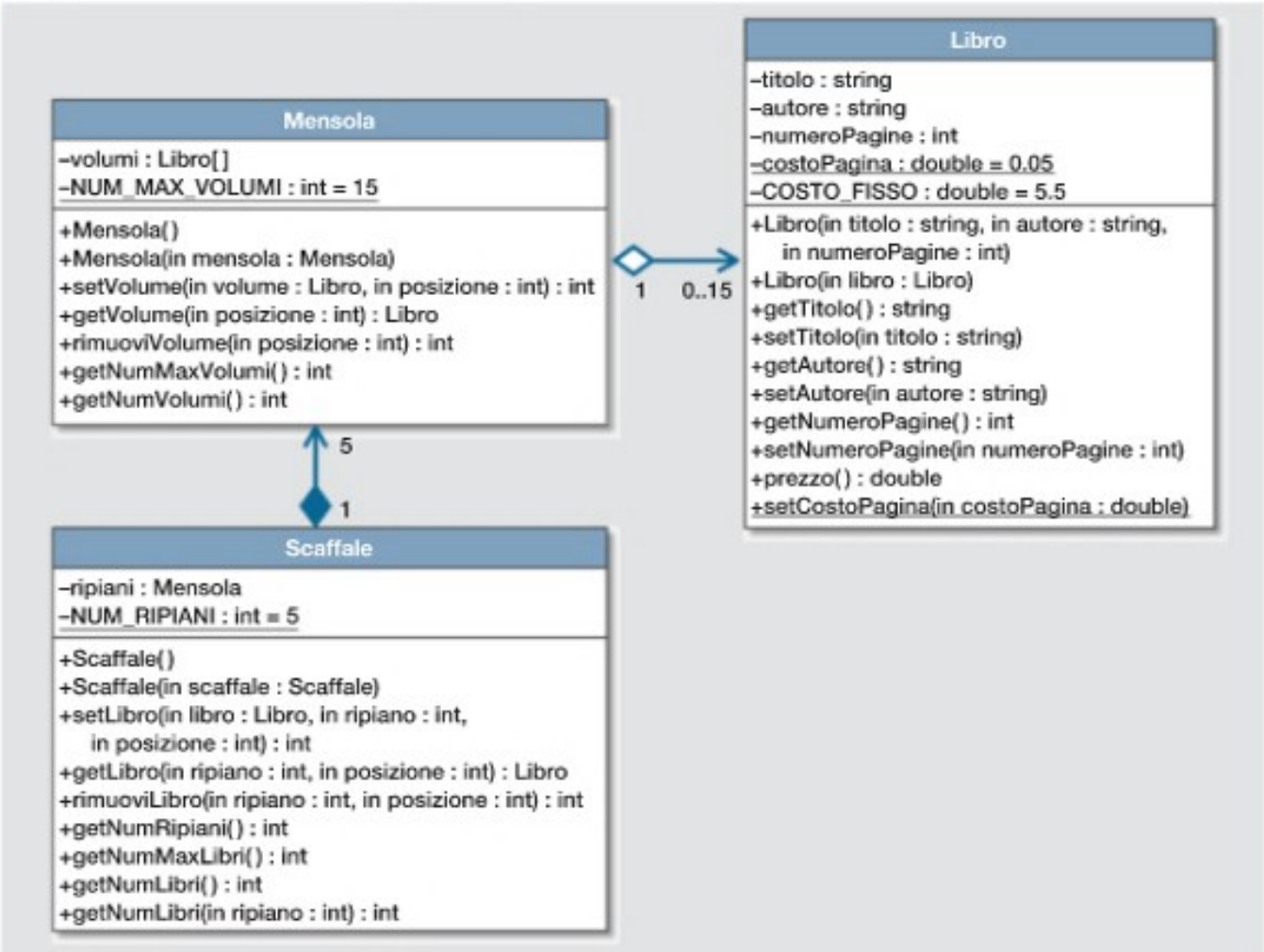


FIGURA 6

Uno scaffale non può esistere senza le mensole che lo costituiscono; per questa ragione l'associazione tra le classi *Scaffale* e *Mensola* è di tipo compositivo. Una possibile implementazione Java della classe *Scaffale* è la seguente:

```
public class Scaffale {
  // attributi
  private static final int NUM_RIPIANI=5;
  private Mensola ripiani[];

  // costruttori
  public Scaffale() {
    int ripiano;

    ripiani=new Mensola[NUM_RIPIANI];
    for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
      ripiani[ripiano]=new Mensola();
    }
  }

  public Scaffale(Scaffale scaffale) {
    int ripiano, posizione;
    Libro libro;

    ripiani=new Mensola[NUM_RIPIANI];
    for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
      ripiani[ripiano]=new Mensola();
      for (posizione=0; posizione<ripiani[ripiano].getNumMaxVolumi(); ripiano++) {
      libro=scaffale.getLibro(ripiano, posizione);
      if (libro!= null)
        ripiani[ripiano].setVolume(libro, posizione);
      }
    }
  }

  // metodi
  public Libro getLibro(int ripiano, int posizione) {
    if ((ripiano<0)||(ripiano>=NUM_RIPIANI))
      return null; // ripiano non valido

    return ripiani[ripiano].getVolume(posizione);
  }

  public int setLibro(Libro libro, int ripiano, int posizione) {
    if ((ripiano<0)||(ripiano>=NUM_RIPIANI))
      return -1; // ripiano non valido

    if (ripiani[ripiano].setVolume(libro, posizione)<0)
      return -2; // posizione nel ripiano non valida o non vuota

    return 1; // inserimento effettuato
  }

  public int rimuoviLibro(int ripiano, int posizione) {
    if ((ripiano<0)||(ripiano>=NUM_RIPIANI))
      return -1; // ripiano non valido
```

```
      if (ripiani[ripiano].rimuoviVolume(posizione)<0)
        return -2; // posizione nel ripiano non valida o vuota

      return 1; // eliminazione effettuata
    }

    public int getNumRipiani() {
      return NUM_RIPIANI;
    }

    public int getNumMaxLibri() {
      int ripiano, n=0;

      for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
        n += ripiani[ripiano].getNumMaxVolumi();
      }
      return n;
    }

    public int getNumLibri() {
      int ripiano, n=0;

      for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
        n += ripiani[ripiano].getNumVolumi();
      }
      return n;
    }

    public int getNumLibri(int ripiano) {
      if ((ripiano<0)||(ripiano>=NUM_RIPIANI))
        return -1; // ripiano non valido

      return ripiani[ripiano].getNumVolumi();
    }
  }
```

OSSERVAZIONE Nell'esempio precedente il codice dei costruttori, oltre a inizializzare il vettore *ripiani*, provvede a invocare esplicitamente il costruttore di ogni singola mensola che costituisce un ripiano per evitare che il vettore *ripiani* abbia elementi nulli. Questo è uno dei motivi per cui le specifiche *Javabeans* prevedono che ogni classe abbia un costruttore di default privo di parametri.

In Java gli *array* hanno un attributo specifico, denominato *length*, che indica il numero di elementi; di conseguenza uno qualsiasi dei cicli dell'esempio precedente potrebbe essere così riformulato:

```
for (ripiano=0; ripiano<ripiani.length; ripiano++)
{
  ...
  ...
  ...
}
```

L'esistenza dell'attributo *length* per gli oggetti di tipo *array* consente il passaggio di un *array* come parametro di un metodo senza la necessità di specificarne la dimensione mediante un diverso parametro, com'è solito fare in C/C++.

ESEMPIO

Un test parziale della classe *Scaffale* può essere eseguito mediante il seguente metodo *main*:

```
public static void main (String[] args) {
   Scaffale scaffale = new Scaffale();
   Libro libro;

   // creazione di tre oggetti di tipo Libro
   Libro l1=new Libro("Pinocchio", "C. Collodi", 150);
   Libro l2=new Libro("Pollicino", "C. Perrault", 80);
   Libro l3=new Libro("La bella addormentata nel bosco", "C. Perrault", 50);

   // inserimento mensola #0
   scaffale.setLibro(l1, 0, 10);
   scaffale.setLibro(l2, 0, 0);
   // test inserimento mensola #1
   libro=new Libro("Cappuccetto Rosso", "F.lli Grimm", 150);
   scaffale.setLibro(libro, 1, 1);

   // test errori inserimento
   if (scaffale.setLibro(l3, 10, 0) == -1)
     System.out.println("mensola non valida");
   if (scaffale.setLibro(l3, 0, 20) == -2)
     System.out.println("posizione non valida o non libera");
   if (scaffale.setLibro(l3, 0, 10) == -2)
     System.out.println("posizione non valida o non libera");

   // inserimento mensola #1
   scaffale.setLibro(l3, 1, 0);

   // visualizzazione contenuto mensole
   for (int ripiano=0; ripiano<scaffale.getNumRipiani(); ripiano++) {
     for (int posizione=0; posizione<scaffale.ripiani[ripiano].getNumMaxVolumi();
          posizione++) {
       libro = scaffale.getLibro(ripiano, posizione);
       if (libro != null)
         System.out.println("ripiano: "+ripiano+" posizione: "+posizione+" -> "+
                            libro.getTitolo()+" "+libro.prezzo()+"€");
     }
   }

   // modifica titolo e autore libro estratto da scaffale
   libro=scaffale.getLibro(0,0);
   if (libro != null) {
     libro.setTitolo("Sussi e Biribissi");
     libro.setAutore("Collodi nipote");
   }

   // visualizzazione contenuto mensole
   for (int ripiano=0; ripiano<scaffale.getNumRipiani(); ripiano++) {
```

```
    for (int posizione=0; posizione<scaffale.ripiani[ripiano].getNumMaxVolumi();
         posizione++) {
      libro = scaffale.getLibro(ripiano, posizione);
      if (libro != null)
        System.out.println("ripiano: "+ripiano+" posizione: "+posizione+" -> "+
                           libro.getTitolo()+" "+libro.prezzo()+"€");
    }
  }
}
```

il cui output è:

```
mensola non valida
posizione non valida o non libera
posizione non valida o non libera
ripiano: 0 posizione: 0 -> Pollicino 9.5€
ripiano: 0 posizione: 10 -> Pinocchio 13.0€
ripiano: 1 posizione: 0 -> La bella addormentata nel bosco 8.0€
ripiano: 1 posizione: 1 -> Cappuccetto Rosso 13.0€
ripiano: 0 posizione: 0 -> Sussi e Biribissi 9.5€
ripiano: 0 posizione: 10 -> Pinocchio 13.0€
ripiano: 1 posizione: 0 -> La bella addormentata nel bosco 8.0€
ripiano: 1 posizione: 1 -> Cappuccetto Rosso 13.0€
```

Nell'esempio precedente gli attributi dei libri contenuti in un oggetto di tipo *Scaffale* possono essere alterati modificando gli attributi di un oggetto restituito dal metodo *getLibro*. Questo comportamento è conseguenza del fatto che il metodo *getLibro* restituisce un riferimento allo stesso oggetto di classe *Libro* riferito dall'oggetto *Scaffale* stesso e non è il comportamento atteso dal programmatore: rappresenta una situazione che, come vedremo più avanti, dovrebbe essere evitata.

2 Oggetti e riferimenti: implementazione e uso del costruttore di copia

Le variabili di tipi di dato non primitivi del linguaggio Java contengono, anziché un valore, un riferimento a un oggetto nell'area di memoria riservata (*heap*). Un **costruttore di copia** è un metodo costruttore che consente di istanziare un nuovo oggetto come copia di un oggetto creato in precedenza e fornito come parametro. In pratica il codice di un costruttore di copia non fa altro che assegnare agli attributi dell'oggetto in fase di creazione il valore degli omonimi attributi dell'oggetto passato come parametro. In questo modo, per esempio, opera il costruttore di copia della classe *Libro* del paragrafo precedente:

```
public Libro(Libro libro) {
  this.titolo=libro.getTitolo();
```

```
  this.autore=libro.getAutore();
  this.numeroPagine=libro.getNumeroPagine();
}
```

Quando gli attributi di una classe non sono di un tipo di dato primitivo o di tipo *String*[1], ma sono riferimenti a oggetti creati istanziando una classe, la situazione richiede un po' di attenzione.

1. Il fatto che gli oggetti di classe *String* siano immutabili li rende, da questo punto di vista, utilizzabili come variabili di un tipo di dato primitivo.

ESEMPIO

Il costruttore di copia della classe *Mensola* introdotta nei paragrafi precedenti

```
public Mensola(Mensola mensola) {
  int posizione;

  volumi=new Libro[MAX_NUM_VOLUMI];
  for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++) {
    if (mensola.getVolume(posizione)!= null)
      volumi[posizione] = mensola.getVolume(posizione);
  }
```

presenta un codice formalmente corretto che si limita ad assegnare ai vari elementi dell'attributo *volumi* dell'oggetto *m* di classe *Mensola* in fase di costruzione gli stessi valori dell'oggetto *mensola* passato come parametro al costruttore.
Quello che il costruttore copia sono però dei **riferimenti** e non dei valori: i riferimenti presenti nell'attributo *volumi* dell'oggetto in fase di costruzione saranno gli stessi riferimenti dell'oggetto *mensola*.
Eseguendo il seguente frammento di codice

```
Libro l1 = new Libro("Pinocchio", "C. Collodi", 150);
Libro l2 = new Libro("Pollicino", "C. Perrault", 80);
Mensola m1 = new Mensola();
m1.setVolume(l1, 0);
m1.setVolume(l2, 2);
Mensola m2 = new Mensola(m1);
```

la situazione che si ottiene è quella presentata in FIGURA 7.

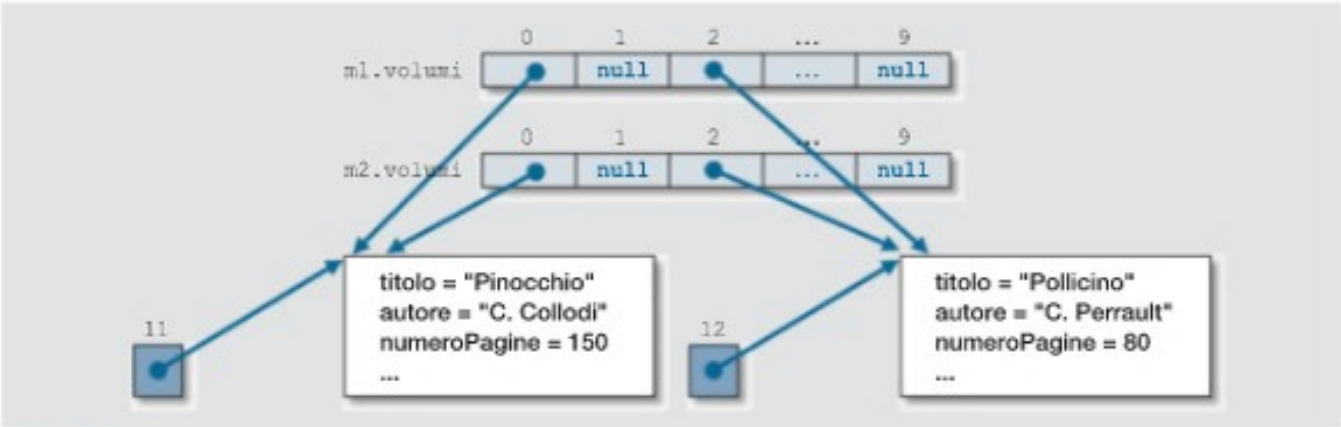


FIGURA 7

Come conseguenza si ha che, apportando delle modifiche agli attributi di un libro contenuto nell'oggetto di classe *Mensola m1*, si modifica anche il corrispondente libro contenuto nell'oggetto di classe *Mensola m2* (e viceversa), perché di fatto sono un solo oggetto e, a meno che la situazione non sia esplicitamente voluta dal programmatore, potrebbero ingenerarsi ambiguità. L'esecuzione della seguente istruzione

```
m1.getVolume(2).setTitolo("La bella addormentata nel bosco");
```

modifica l'oggetto *l2* e, di conseguenza, il contenuto informativo degli oggetti riferiti da entrambe le mensole *m1* ed *m2* (FIGURA 8). ▶

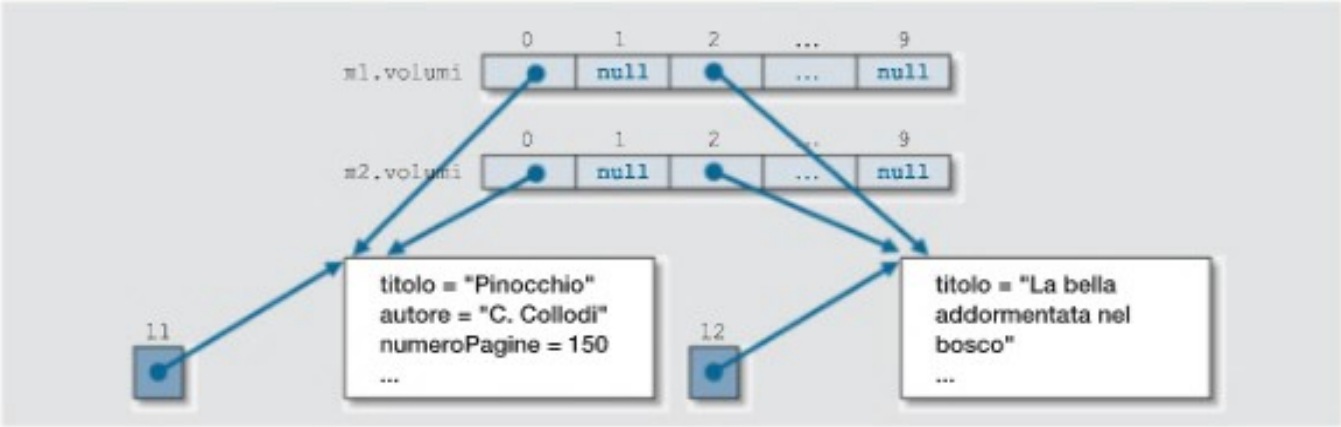


FIGURA **8**

La versione corretta del costruttore di copia crea invece un nuovo oggetto istanza della classe *Mensola* con l'attributo *volumi*, che riferisce nuovi oggetti di tipo *Libro*, «clonati» a partire dagli oggetti di tipo *Libro* riferiti dall'attributo *volumi* dell'oggetto di tipo *Mensola* fornito come parametro

```
public Mensola(Mensola mensola) {
  int posizione;

  volumi=new Libro[MAX_NUM_VOLUMI];
  for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++) {
    if (mensola.getVolume(posizione)!= null)
      volumi[posizione] = new Libro(mensola.getVolume(posizione));
}
```

per cui l'esecuzione del frammento di codice

```
Libro l1 = new Libro("Pinocchio", "C. Collodi", 150);
Libro l2 = new Libro("Pollicino", "C. Perrault", 80);
Mensola m1 = new Mensola();
m1.setVolume(l1, 0);
m1.setVolume(l2, 2);
Mensola m2 = new Mensola(m1);
```

dà origine ora alla situazione di FIGURA **9**. Gli oggetti *m1* ed *m2* hanno ora copie separate dei propri attributi ed eventuali modifiche apportate a uno dei due oggetti sono ininfluenti per l'altro.

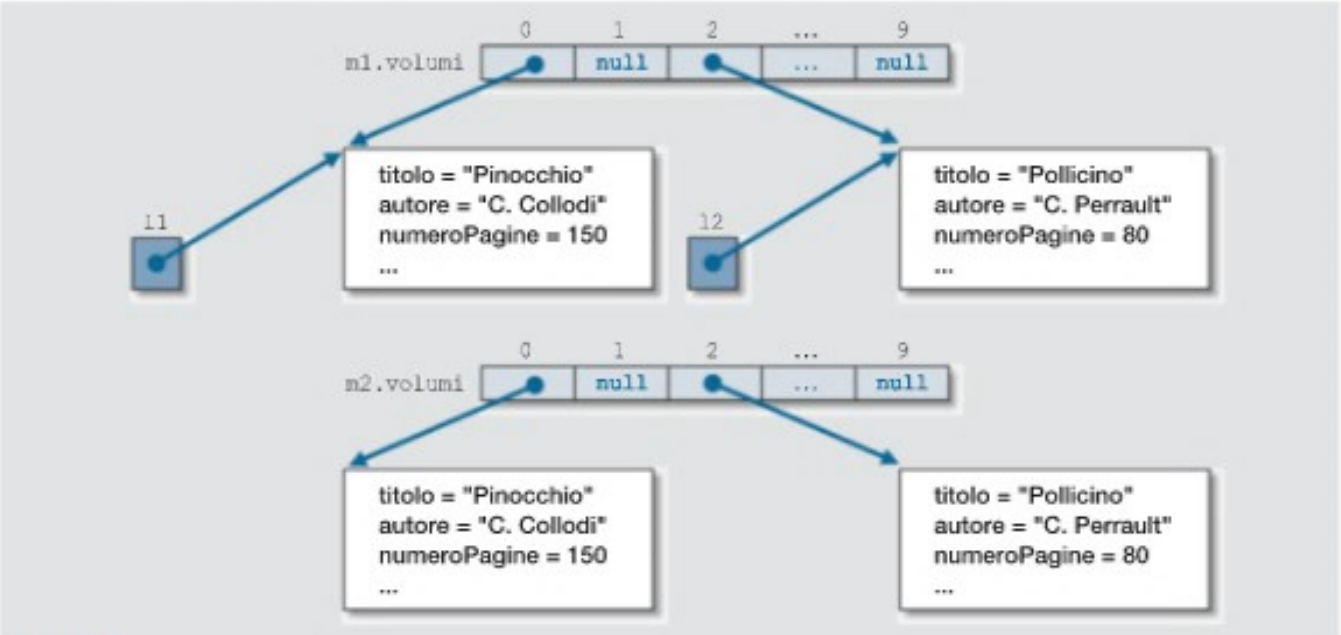


FIGURA **9**

In realtà il problema illustrato nell'esempio precedente è propagato anche da altri metodi della classe *Mensola*. Per esempio il metodo *getVolume*

```
public Libro getVolume(int posizione) {
  if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
    return null; // posizione non valida: nessun libro
                 // restituito
  return volumi[posizione]; // restituisce il libro
                            // in posizione
}
```

restituisce lo stesso riferimento a un oggetto di tipo *Libro* presente nell'*array volumi* attributo dell'oggetto di classe *Mensola* su cui viene invocato. Anche in questo caso una modifica dell'oggetto restituito esternamente alla classe si riflette sul contenuto dell'oggetto interno alla classe, probabilmente contro l'intenzione del programmatore.

Un modo più sicuro di implementare il metodo *getVolume* è il seguente, che sfrutta il costruttore di copia della classe *Libro* per restituire un oggetto di tipo *Libro* clonato da quello riferito dall'oggetto di classe *Mensola*:

```
public Libro getVolume(int posizione) {
  if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
    return null; // posizione non valida: nessun libro
                 // restituito
// restituisce una copia del libro in posizione
  if (volumi[posizione]!= null)
    return new Libro(volumi[posizione]);
  else
    return null;
}
```

Non potendo clonare un oggetto `null`, diviene indispensabile trattare separatamente questo caso.

In modo meno evidente anche il metodo *setVolume* presenta lo stesso problema:

```
public int setVolume(Libro libro, int posizione) {
  if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
    return -1; // posizione non valida
  if (volumi[posizione]!= null)
    return -2; // posizione occupata
  volumi[posizione]=libro; // inserimento libro in posizione
  return posizione; // restituisce la posizione di inserimento
}
```

Infatti il codice esterno alla classe *Mensola* che lo invoca mantiene un riferimento all'oggetto di tipo *Libro* fornito come parametro e, trattandosi dello stesso riferimento che viene copiato nell'elemento dell'*array volumi*, una sua eventuale successiva modifica si rifletterebbe sul contenuto informativo dell'oggetto istanza della classe *Mensola*. Anche in questo caso è semplice porre rimedio utilizzando opportunamente il costruttore di copia della classe *Libro* per clonare l'oggetto prima di riferirlo:

```
public int setVolume(Libro libro, int posizione) {
  if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
    return -1; // posizione non valida
  if (volumi[posizione]!= null)
    return -2; // posizione occupata
  // inserimento copia di libro in posizione
  volumi[posizione]=new Libro(libro);
  return posizione; // restituisce la posizione di inserimento
}
```

ESEMPIO

Tenendo conto delle considerazioni fatte, il seguente codice Java implementa una versione corretta della classe *Mensola*:

```
public class Mensola {
  // attributi
  private static final int MAX_NUM_VOLUMI=15;
  private Libro volumi[];

  // costruttori
  public Mensola() {
    volumi=new Libro[MAX_NUM_VOLUMI];
  }

  public Mensola(Mensola mensola) {
    int posizione;

    volumi=new Libro[MAX_NUM_VOLUMI];
    for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++) {
      if (mensola.getVolume(posizione)!= null)
        volumi[posizione]=new Libro(mensola.getVolume(posizione));
    }
  }

  // metodi
  public int setVolume(Libro libro, int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return -1; // posizione non valida
    if (volumi[posizione]!= null)
      return -2; // posizione occupata
    // inserimento copia di libro in posizione
    volumi[posizione]=new Libro(libro);
    return posizione; // restituisce la posizione di inserimento
  }

  public Libro getVolume(int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
      return null; // posizione non valida: nessun libro restituito
    // restituisce una copia del libro in posizione
    if (volumi[posizione]!= null)
      return new Libro(volumi[posizione]);
    else
      return null;
  }

 public int rimuoviVolume(int posizione) {
    if ((posizione<0)||(posizione>=MAX_NUM_VOLUMI))
```

```
        return -1; // posizione non valida
      if (volumi[posizione]== null)
        return -2; // posizione vuota
      volumi[posizione]=null; // rimozione libro in posizione
      return posizione; // restituisce la posizione liberata
    }

    public int getNumMaxVolumi() {
      return MAX_NUM_VOLUMI;
    }

    public int getNumVolumi() {
      int posizione, n=0;
      for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
        if (volumi[posizione]!= null)
          n++;
      return n;
    }
  }
```

OSSERVAZIONE La classe *Scaffale* non ha bisogno di alcun intervento di adeguamento: le modifiche apportate ai metodi *getVolume* e *setVolume* della classe *Mensola*, richiamati dal codice dei metodi della classe *Scaffale*, la rendono sicura rispetto al problema evidenziato della mancata clonazione degli oggetti riferiti internamente.

3 *Array* come parametri e valori di ritorno dei metodi di una classe

Il linguaggio di programmazione Java consente sia di passare un *array* come parametro a un metodo sia di ottenerlo come valore di ritorno.

ESEMPIO Per ottenere la lista dei titoli dei libri di un certo autore presenti su uno scaffale in forma di *array* di stringhe si può predisporre un metodo che svolge le seguenti operazioni:

- conta quanti sono i libri dell'autore analizzando tutte le posizioni di tutte le mensole dello scaffale;
- dimensiona di conseguenza un vettore di stringhe per memorizzare i titoli dei libri dell'autore;
- restituisce il vettore creato, oppure **null** se non è stato trovato alcun libro dell'autore specificato.

Nel linguaggio di programmazione Java il codice del metodo potrebbe essere il seguente:

```
public String[] elencoTitoliAutore(String autore) {
  int ripiano, posizione;
  int libriAutore=0;
  Libro libro;

  for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++)
    for (posizione=0; posizione<ripiani[ripiano].getNumMaxVolumi();
         posizione++)
      if (ripiani[ripiano].getVolume(posizione) != null) {
        libro = ripiani[ripiano].getVolume(posizione);
        if (libro.getAutore().equalsIgnoreCase(autore))
          libriAutore++;
        }
```

```
  if (libriAutore==0)
    return null;
  // dichiarazione e creazione array risultato
  String elencoLibri[] = new String[libriAutore];
  for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++)
    for (posizione=0; posizione<ripiani[ripiano].getNumMaxVolumi();
         posizione++)
      if (ripiani[ripiano].getVolume(posizione) != null) {
        libro = ripiani[ripiano].getVolume(posizione);
          if (libro.getAutore().equalsIgnoreCase(autore)) {
            elencoLibri[libriAutore] = libro.getTitolo();
          }
      }
  return elencoLibri;
}
```

Questo metodo può essere verificato nel metodo *main* con le seguenti istruzioni

```
...
String titoli[]=scaffale.elencoTitoliAutore("C. Perrault");
if (titoli.length>0) {
  System.out.println("Libri di C. Perrault sullo scaffale:");
  for (int indice=0; indice<titoli.length; indice++)
    System.out.println(titoli[indice]);
}
else
  System.out.println("Nessun libro di C. Perrault sullo scaffale");
...
```

che producono un output del tipo:

```
Libri di C. Perrault sullo scaffale:
Pollicino
La bella addormentata
```

ESEMPIO

Per creare una mensola su cui disporre già inizialmente alcuni libri è possibile passare al costruttore come parametro un *array* di libri:

- se l'*array* passato come argomento contiene più libri della dimensione della mensola i libri in eccesso sono ignorati;
- se l'*array* non contiene alcun libro viene creata una mensola vuota.

```
public Mensola(Libro[] elencoLibri) {
  int posizione, libri;

  volumi=new Libro[MAX_NUM_VOLUMI];
  if (MAX_NUM_VOLUMI>elencoLibri.length)
    libri = elencoLibri.length;
  else
    libri = MAX_NUM_VOLUMI;
  for (posizione=0; posizione<libri; posizione++)
    volumi[posizione] = new Libro(elencoLibri[posizione]);
  }
```

Questo metodo può essere verificato nel metodo *main* con le seguenti istruzioni:

```
...
Libro elencoLibri[] = new Libro[3];
elencoLibri[0]=new Libro("Pinocchio","C. Collodi", 150);
elencoLibri[1]=new Libro("Pollicino","C. Perrault",80);
elencoLibri[2]=new Libro("La bella addormentata","C. Perrault", 50);
Mensola mensola = new Mensola(elencoLibri);

// visualizzazione elenco volumi
for (int posizione=0; posizione<mensola.getNumMaxVolumi();
     posizione++) {
  Libro libro = mensola.getVolume(posizione);
  if (libro != null)
    System.out.println("posizione "+posizione+" -> "+
                       libro.getTitolo()+" "+libro.prezzo()+"€");...
...
```

Si ottiene in output:

```
posizione 0 -> Pinocchio 13.0€
posizione 1 -> Pollicino 9.5€
posizione 2 -> La bella addormentata 8.0€
```

4 Eccezioni predefinite non controllate

Le **eccezioni** sono eventi che si presentano in fase di esecuzione (*run-time*) di un programma e sono generalmente collegate al verificarsi di situazioni anomale.

Tra le eccezioni più comuni segnaliamo: la divisione per zero, l'uso di un indice fuori dal *range* di un *array*, l'accesso a un riferimento nullo perché non inizializzato.

Al verificarsi di un'eccezione si hanno due possibilità: intercettarla e gestirla mediante codice specifico, oppure lasciare che il programma termini la sua esecuzione in maniera anomala e con conseguenze imprevedibili rispetto all'elaborazione interrotta.

Le cause che possono portare alla generazione di un'eccezione sono molteplici; classifichiamo nel seguito le più comuni:

- **errori software**: errori imputabili allo sviluppatore;
- **input errato da parte dell'utente**: l'immissione di dati in un formato non consentito è una delle più comuni cause di eccezione nell'esecuzione di un programma; la responsabilità è comunque dello sviluppatore che avrebbe dovuto prevedere nel codice controlli per prevenire la situazione di errore;
- **violazioni della sicurezza**: un tipico esempio è il tentativo di modificare un file senza avere i necessari diritti per farlo;

Eccezioni in Java e C++

In Java la gestione delle eccezioni (*exception-handling*) consiste in più costrutti del linguaggio finalizzati a rendere semplice, chiara e sicura la gestione delle situazioni anomale che si possono verificare durante l'esecuzione di un programma.
L'*exception-handling* di Java deriva direttamente da quello del linguaggio C++; è forse più oneroso, ma certamente più sicuro. Realizza infatti la regola «*handle or declare*» («gestisci o dichiara»), che obbliga il programmatore a prevedere esplicite contromisure per ogni situazione anomala potenziale.

- **indisponibilità delle risorse**: errore dovuto alla non disponibilità temporanea o permanente di una risorsa hardware (per esempio una connessione di rete) o software (per esempio un file).

Quando si verifica un'eccezione non gestita dal codice del programma, la JVM interrompe l'esecuzione del programma e produce un messaggio noto come ***stack-trace*** che riporta, oltre al tipo di eccezione generata e il metodo che l'ha causata, la catena di invocazione dei metodi che hanno determinato l'eccezione stessa.

ESEMPIO

Il metodo *main* della seguente classe Java ha il solo scopo di generare un'eccezione di divisione per zero:

```
1 public class Eccezione {
2
3   static int divisione(int a, int b) {
4     return a/b;
5   }
6
7   static int quoziente10(int d) {
8     return divisione(10, d);
9   }
10
11  public static void main (String[] args) {
12    for (int n=10; n>=0; n--)
13      System.out.println(quoziente10(n));
14  }
15 }
```

Il messaggio di output prodotto al verificarsi dell'eccezione (quando la variabile *n* assume valore 0) specifica la catena di invocazione del metodo che ha causato l'eccezione stessa; in questo caso il metodo *divisione* invocato dal metodo *quoziente10*, a sua volta invocato dal metodo *main*:

```
1
1
1
1
1
2
2
3
5
10
Exception in thread "main" java.lang.ArithmeticException:
  / by zero
  at Eccezione.divisione(Eccezione.java:4)
  at Eccezione.quoziente10(Eccezione.java:8)
  at Eccezione.main(Eccezione.java:13)
```

Naturalmente l'interruzione dell'esecuzione di un programma a causa di un'eccezione non è un evento auspicato; per questa ragione Java – come altri linguaggi di programmazione – prevede dei meccanismi per la gestione di tali eventi finalizzati alla costruzione di programmi robusti e affidabili.

Anche se il tipo di errore che ha generato un'eccezione non è recuperabile, nel senso che il programma dovrà essere in ogni caso terminato, potrebbe essere comunque importante intercettare l'eccezione e gestire correttamente la terminazione del programma per rilasciare in modo controllato le eventuali risorse acquisite (per esempio per chiudere correttamente un file aperto in scrittura evitando una possibile perdita di dati).

Nel linguaggio di programmazione Java le eccezioni sono esse stesse delle classi (contenute nel *package* predefinito *java.lang*) che derivano dalla classe *Exception*, che deriva a sua volta dalla classe *Throwable*.

Mediante il meccanismo dell'ereditarietà il linguaggio Java raggruppa in categorie le eccezioni predefinite di cui si elencano le più comuni:

- *ArithmeticException*: errori aritmetici;
- *NullPointerException*: uso di riferimenti nulli non inizializzati;
- *ArrayIndexOutOfBoundsException*: indicizzazione di un *array* fuori dai limiti;
- *IOException*: errori di I/O; ...

Le eccezioni predefinite del linguaggio Java sono di tipo ***unchecked*** («non controllato»): lo sviluppatore ha la facoltà di intercettarle per gestirle, o di non intercettarle affatto lasciando che la loro eventuale generazione porti a una terminazione anomala del programma.

L'intercettazione delle eccezioni viene realizzata in Java mediante un costrutto di programmazione ereditato dal linguaggio C++ che costituisce una «trappola» per gli errori:

```
try {
  // istruzioni che potenzialmente generano eccezioni
  ...;
  ...;
  ...;
}
catch (TipoEccezione1 identificatore) {
  // istruzioni da eseguire se si verifica l'eccezione
  // TipoEccezione1
  ...;
  ...;
  ...;
}
catch (TipoEccezione2 identificatore) {
  // istruzioni da eseguire se si verifica l'eccezione
  // TipoEccezione2
  ...;
  ...;
  ...;
}
finally {
  // istruzioni da eseguire per un qualsiasi tipo di eccezione
  ...;
```

```
  ...;
  ...;
}
```

Se nel corso dell'esecuzione delle istruzioni contenute nel blocco **try** si verificano delle eccezioni, il controllo dell'esecuzione passa alle istruzioni del blocco **catch** corrispondente al tipo di eccezione rilevata: il codice del blocco **catch**, per quanto possibile, dovrebbe ripristinare uno stato corretto. Il programmatore specificherà i blocchi **catch** necessari a catturare i tipi di eccezione che ritiene necessario intercettare. La clausola **finally** è opzionale: essa è obbligatoria solo nel caso in cui non esista alcuna clausola **catch**; se è presente, il blocco **finally** viene sempre eseguito dopo il verificarsi di un'eccezione anche nel caso che il blocco **catch** eseguito contenga un'istruzione **return**, o nel caso che nessun blocco **catch** corrisponda al tipo di eccezione generata.

ESEMPIO

La seguente classe modifica quella dell'esempio precedente prevedendo l'intercettazione e la gestione dell'eccezione dovuta alla divisione per zero; a questo scopo i metodi *divisione* e *quoziente10* restituiscono una stringa di caratteri invece che un valore numerico.

```
1 public class Eccezione {
2
3   static String divisione(int a, int b) {
4     try{
5       return Integer.toString(a/b);
6     }
7     catch(ArithmeticException exception) {
8       return "impossibile calcolare "+a+"/"+b;
9       }
10    }
11
12  static String quoziente10(int d) {
13    return divisione(10, d);
14  }
15
16  public static void main (String[] args) {
17    for (int n=10; n>=0; n--)
18      System.out.println(quoziente10(n));
19  }
20 }
```

L'output che si ottiene è il seguente e il programma non termina in questo caso l'esecuzione in modo anomalo:

```
1
1
1
1
1
2
2
3
5
10
impossibile calcolare 10/0
```

Infatti, al verificarsi dell'eccezione di classe *ArithmeticException* (*exception* è un oggetto istanza di questa classe specifico per l'errore rilevato), questa viene intercettata e il codice del blocco **catch** costruisce in questo caso il messaggio di risposta appropriato.

I possibili tipi di eccezioni generate in un programma Java sono strutturati in una gerarchia di classi rappresentata schematicamente e in maniera non esaustiva nel diagramma UML di FIGURA **10**.

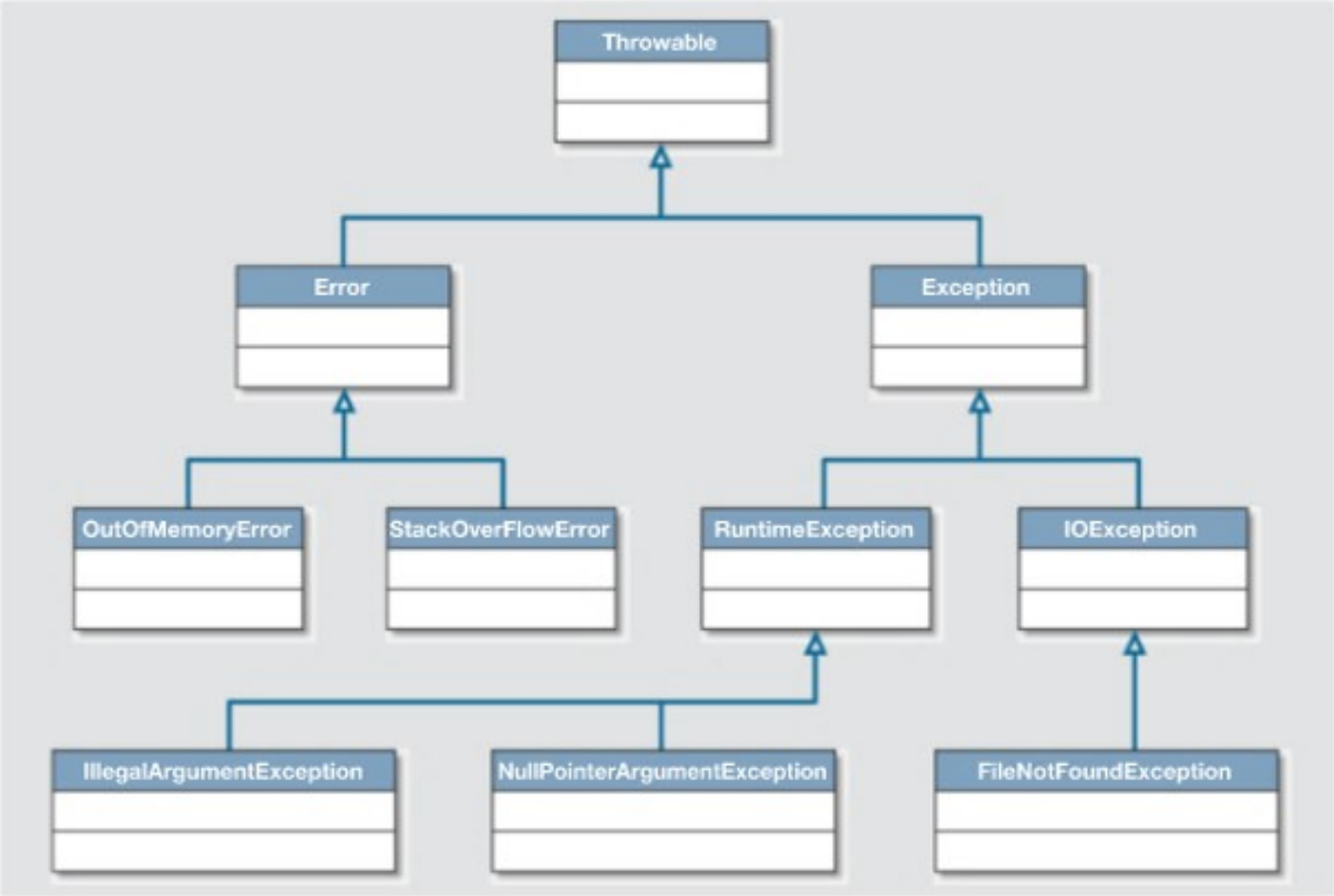


FIGURA **10**

Le eccezioni che derivano dalla classe *Error* (generalmente relative al funzionamento della JVM), pur essendo intercettabili, sono in ogni caso irrecuperabili. Le eccezioni che derivano dalla classe *RuntimeException* sono eccezioni *unchecked*, che il programmatore può decidere di intercettare o meno; le altre eccezioni che derivano dalla classe *Exception* sono di tipo *checked* («controllato») e il programmatore deve necessariamente intercettarle e gestirle.

ESEMPIO La classe *Mensola* introdotta in precedenza può essere riscritta utilizzando opportunamente l'intercettazione delle eccezioni per evitare i tediosi controlli sulla correttezza degli indici dei vettori e sulla nullità o meno dei riferimenti:

```
public class Mensola {
   // attributi
   private static final int MAX_NUM_VOLUMI=15;
   private Libro volumi[];

   // costruttori
   public Mensola() {
      volumi=new Libro[MAX_NUM_VOLUMI];
   }

public Mensola(Mensola mensola) {
      int posizione;
```

```
    volumi=new Libro[MAX_NUM_VOLUMI];
    for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++) {
      if (mensola.getVolume(posizione)!= null)
        volumi[posizione]=new Libro(mensola.getVolume(posizione));
    }
  }

  // metodi
  public int setVolume(Libro libro, int posizione) {
    try {
        // inserimento copia di libro in posizione
        if (volumi[posizione]!= null)
          return -2; // posizione occupata
        volumi[posizione]=new Libro(libro);
        return posizione; // restituisce la posizione di inserimento
    }
    catch (ArrayIndexOutOfBoundsException exception) {
      return -1; // posizione non valida
    }
  }

  public Libro getVolume(int posizione) {
    try {
        // restituisce una copia del libro in posizione
        return new Libro(volumi[posizione]);
    }
    catch (ArrayIndexOutOfBoundsException exception) {
      return null; // posizione non valida: nessun libro restituito
    }
    catch (NullPointerException exception) {
      return null; // posizione vuota: nessun libro restituito
    }
  }

  public int rimuoviVolume(int posizione) {
    try {
        volumi[posizione]=null; // rimozione libro in posizione
        return posizione; // restituisce la posizione liberata
    }
    catch (ArrayIndexOutOfBoundsException exception) {
      return -1; // posizione non valida
    }
  }

  public int getNumMaxVolumi() {
    return MAX_NUM_VOLUMI;
  }

  public int getNumVolumi() {
    int posizione, n=0;
    for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
      if (volumi[posizione]!= null)
        n++;
    return n;
  }
}
```

Essendo un'eccezione Java un normale oggetto (istanza di una classe che deriva dalla classe *Exception*), è possibile da parte del programmatore definire nuovi tipi di eccezione definendo nuove classi che derivano dalla classe *Exception*.

5 Definizione e generazione delle eccezioni

Ogni singolo metodo di una classe viene progettato e implementato per effettuare una specifica operazione: in presenza di situazioni anomale è possibile che un metodo fallisca senza portare a termine l'operazione.

Questa evenienza deve essere segnalata al metodo invocante che potrà, a seconda dei casi, prevedere o meno una contromisura per concludere il proprio compito nonostante il fallimento del metodo chiamato, oppure, se questo risulta impossibile, segnalare a sua volta il proprio fallimento nei confronti del proprio chiamante.

Per segnalare il proprio insuccesso un metodo Java può generare (si utilizzano comunemente anche i verbi «sollevare» o «lanciare») un'eccezione. La generazione di un'eccezione da parte di un metodo è un comportamento analogo alla restituzione di un valore, ma in Java i due concetti sono volutamente distinti: per esempio un metodo può restituire un valore di tipo `int`, oppure generare un'eccezione che è un oggetto istanza di una classe che deriva dalla classe *Exception*.

Un metodo o un costruttore che può generare un'eccezione deve specificare questa eventualità nella propria firma utilizzando la parola chiave `throws` seguita dal tipo di eccezione, o da un elenco di tipi di eccezioni, che possono essere sollevate dal metodo. Nel codice del metodo o del costruttore la parola chiave `throw` consente di interrompere l'esecuzione e di sollevare l'eccezione specificata come un nuovo oggetto.

ESEMPIO La dichiarazione del metodo *differenzaOrari* di un'ipotetica classe *GestioneOrari* specifica che possono essere sollevate eccezioni di tipo *OrarioNonValido*, la cui effettiva generazione avviene se i valori delle ore (0-23), dei minuti (0-59) e dei secondi (0-59) forniti come parametri non rispettano i limiti verificati dal metodo privato *orarioValido* (si noti come il membro che si trova a destra del `return` di quest'ultimo metodo, essendo un'espressione logica, possa restituire i soli valori `true` o `false`):

```
/* verifica la validità di un orario espresso nel formato h:m:s (ore, minuti e secondi)
*/
private static boolean orarioValido(int h, int m, int s) {
  return (h>=0 && h<24 && m>=0 && m<60 && s>=0 && s<60);
}

/* calcola la differenza in secondi fra 2 orari espressi nel formato
   h:m:s (ore, minuti e secondi)
*/
public static int differenzaOrari(int h1, int m1, int s1, int h2, int m2, int s2)
                                  throws OrarioNonValido {
if (!orarioValido(h1, m1, s1) || !orarioValido(h2, m2, s2))
```

```
    throw new OrarioNonValido();
  else {
        int sec1, sec2;
        sec1 = h1*3600 + m1*60 + s1;
        sec2 = h2*3600 + m2*60 + s2;
        return (sec2-sec1);
      }
}
```

Se non si verificano errori il metodo *differenzaOrari* restituisce un valore intero che rappresenta il numero di secondi che intercorrono tra i due orari. Nel caso in cui uno dei due orari non risulti valido (per esempio 12h, 62m e 54s), invece di restituire un valore intero il metodo esegue l'istruzione

```
throw new OrarioNonValido();
```

che istanzia e solleva un'eccezione di tipo *OrarioNonValido*. Il tipo di eccezione *OrarioNonValido* deve essere stato in precedenza definito come una classe, eventualmente nella seguente forma minimale:

```
public class OrarioNonValido extends Exception {
}
```

La semantica dell'istruzione `throw` ha alcuni aspetti in comune con quella dell'istruzione `return`. In particolare l'esecuzione di `throw` implica la terminazione immediata del metodo e il passaggio del controllo al codice chiamante del metodo stesso.

È importante non confondere la *keyword* `throw` (un'istruzione che genera un'eccezione) con la *keyword* `throws` (usata nella firma di un metodo per elencare i tipi di eccezione che esso può generare).

Il codice che invoca un metodo che dichiara la potenziale generazione di eccezioni utilizzando la parola chiave `throws` nella firma deve gestirle mediante un blocco `try/catch`.

ESEMPIO

Il metodo *main* della classe *GestioneOrari* dell'esempio precedente deve verificare il corretto funzionamento del metodo *differenzaOrari* anche nel caso di orari non validi:

```
public static void main(String[] args) {
   ...
   try {
        int ss = GestioneOrari.differenzaOrari(0, 0, 0, 23, 59, 59);
        System.out.println("Secondi di differenza: "+ss);
   }
   catch(OrarioNonValido exception) {
     System.out.println("Errore nell'orario specificato!");
   }
   try {
        int ss = GestioneOrari.differenzaOrari(0, 0, 0, 24, 0, 0);
        System.out.println("Secondi di differenza: "+ss);
   }
   catch(OrarioNonValido exception) {
     System.out.println("Errore nell'orario specificato!");
   }
   ...
}
```

Il blocco **try** controlla la corretta esecuzione delle istruzioni che lo costituiscono, in particolare in questo caso dell'invocazione del metodo *differenzaOrari*, che può generare due situazioni distinte:

- il metodo ha successo e ritorna un valore intero (è quello che accade nel primo caso): l'esecuzione delle istruzioni del blocco prosegue normalmente e al termine il controllo passa alla prima istruzione successiva al blocco **catch**;
- il metodo fallisce e solleva un'eccezione di tipo *OrarioNonValido* (è quello che accade nel secondo caso): l'esecuzione delle istruzioni si interrompe e sono eseguite le istruzioni del blocco **catch** prima che il controllo passi alla prima istruzione successiva al blocco stesso.

Il blocco **try/catch** consente di separare il funzionamento del metodo nel caso «normale» e nel caso in cui sia necessario gestire eventuali situazioni anomale; il metodo *main* precedente visualizza il seguente output:

```
Secondi di differenza: 86399
Errore nell'orario specificato!
```

Nel caso in cui un metodo non sia in grado di effettuare azioni di recupero di un'eccezione sollevata da un metodo invocato non può che risollevare l'eccezione al metodo che a suo volta lo ha invocato.

ESEMPIO

Se la classe *GestioneOrari* prevede un metodo statico per confrontare due orari tra loro, è ragionevole delegare la gestione di eventuali errori dei parametri forniti al metodo che li ha forniti:

```
public static boolean confrontaOrari(int h1, int m1, int s1, int h2, int m2, int s2)
                                        throws OrarioNonValido {
if (differenzaOrari(h1, m1, s1, h2, m2, s2) == 0)
  return true;
else
  return false;
}
```

Il metodo *confrontaOrari* non prevede un blocco **try/catch**, e di conseguenza non può intercettare l'eccezione eventualmente generata dall'invocazione del metodo *differenzaOrari*. In questo caso l'esecuzione del codice viene interrotta e l'eccezione generata da *differenzaOrari* viene propagata al metodo che ha invocato il metodo *confrontaOrari*, esattamente come se questo avesse eseguito un'istruzione **throw**. Per questo motivo è obbligatorio inserire la dichiarazione

```
throws OrarioNonValido
```

anche nella firma del metodo *confrontaOrari*, segnalando così il fatto che anche questo metodo può indirettamente generare un'eccezione di tipo *OrarioNonValido*.

Nel linguaggio Java, in relazione alla gestione delle eccezioni, vale la cosiddetta regola «***handle or declare***» («gestisci o dichiara»): a fronte di una possibile eccezione un metodo deve gestirla utilizzando un blocco **try/catch** oppure esplicitare a sua volta di sollevarla specificando la clausola **throws**. In definitiva, una potenziale eccezione non deve mai passare inosservata: se non viene intercettata e gestita, deve essere esplicitamente rimandato al chiamante l'obbligo di gestirla.

Le eccezioni dichiarate dal programmatore in linguaggio Java sono normalmente di tipo *checked*, per le quali si applica la regola «*handle or declare*».

OSSERVAZIONE Nei linguaggi che non prevedono la gestione esplicita delle eccezioni, un metodo, o una funzione, segnalano di norma il proprio fallimento ritornando un valore particolare a cui il programmatore

attribuisce convenzionalmente il significato di segnalazione di un errore. In questo contesto, per esempio, il metodo *differenzaOrari* potrebbe restituire il valore negativo –1 in caso di fallimento. Questa tecnica di gestione delle «eccezioni» presenta però diverse controindicazioni:

- la segnalazione non è imposta dal linguaggio, ma segue una convenzione che deve essere documentata accuratamente affinché il metodo chiamante possa interpretare il valore restituito dal metodo invocato;
- non sempre è possibile identificare un valore particolare da usare come segnalazione di errore; se, per esempio, il metodo *differenzaOrari* è pensato per fornire la differenza non in valore assoluto, allora il valore negativo –1 è un risultato lecito e non può rappresentare una condizione di errore anomala.

In ogni caso, senza gestione delle eccezioni non esiste alcun vincolo che imponga al metodo chiamante di verificare se il metodo invocato ha fallito: rendendo obbligatoria la gestione delle eccezioni sollevate, il modello proposto dal linguaggio Java impedisce alle anomalie di esecuzione di passare inosservate. Questo comporta un onere per il programmatore, ma è un importante contributo alla robustezza e all'affidabilità del programma.

Le eccezioni generate negli esempi precedenti sono istanze della classe *OrarioNonValido*:

il linguaggio di programmazione Java impone che le eccezioni siano oggetti istanza di una classe che deriva da *Exception*, o dalla superclasse *Throwable*.

Se si esclude questo vincolo, la definizione di una classe di eccezione è lasciata allo sviluppatore; in particolare è possibile usare attributi e metodi per dotare l'oggetto che rappresenta un'eccezione di informazioni specifiche sul tipo di errore verificatosi.

ESEMPIO La seguente classe, che consente di rappresentare un'eccezione relativa all'orario memorizzando l'orario invalido che ha generato l'eccezione stessa

```
public class OrarioNonValido extends Exception {
   private int h;
   private int m;
   private int s;

   public OrarioNonValido(int h, int m, int s) {
      this.h = h;
      this.m = m;
      this.s = s;
   }

   public int getOre() { return h; }
   public int getMinuti() { return m; }
   public int getSecondi() { return s; }
   public String toString() { return (""+h+":"+m+":"+s); }
}
```

comporta una revisione del metodo statico *differenzaOrari* della classe *GestioneOrari*:

```
public static int differenzaOrari(int h1, int m1, int s1, int h2, int m2, int s2)
                                  throws OrarioNonValido {
  if (!orarioValido(h1, m1, s1))
    throw new OrarioNonValido(h1, m1, s1);
  else if (!orarioValido(h2, m2, s2))
    throw new OrarioNonValido(h2, m2, s2);
  else {
        int sec1, sec2;
        sec1 = h1*3600 + m1*60 + s1;
        sec2 = h2*3600 + m2*60 + s2;
        return (sec2-sec1);
      }
}
```

In questo modo il metodo che intercetta l'anomalia, diversamente dai casi precedenti, dispone di un oggetto che può fornire informazioni sulla natura dell'errore che ha causato l'anomalia stessa e il metodo *main* della classe *GestioneOrari* potrebbe essere il seguente:

```
public static void main(String[] args) {
  ...
  try {
        int ss = GestioneOrari.differenzaOrari(0, 0, 0, 23, 59, 59);
        System.out.println("Secondi di differenza: "+ss);
  }
  catch(OrarioNonValido exception) {
    System.out.println("Errore nell'orario specificato ("+exception.getOre()+":"+
                       exception.getMinuti()+":"+exception.getSecondi()+")!");
  }
  try {
        int ss = GestioneOrari.differenzaOrari(0, 0, 0, 24, 0, 0);
        System.out.println("Secondi di differenza: "+ss);
  }
  catch(OrarioNonValido exception) {
    System.out.println("Errore nell'orario specificato ("+exception.getOre()+":"+
                       exception.getMinuti()+":"+exception.getSecondi()+")!");

  }
  ...
}
```

Il parametro che compare nella clausola **catch** è simile al parametro di un metodo: identifica un riferimento a cui viene associato l'oggetto di tipo eccezione istanziato e generato dall'istruzione **throw**. Tale oggetto può essere manipolato come un qualsiasi altro oggetto e nel secondo caso permette di evidenziare che l'orario 24:00:00 è errato.

ESEMPIO

Per esemplificare quanto esposto effettuiamo un *refactoring* della classe *Scaffale* introdotta nei paragrafi precedenti. Prima di tutto definiamo le classi che rappresentano le eccezioni che i metodi della classe *Scaffale* possono generare

```
public class PosizioneNonValida extends Exception {
  private int posizione;
  private int ripiano;

  public PosizioneNonValida(int posizione, int ripiano) {
    this.posizione = posizione;
```

```
    this.ripiano = ripiano;
  }

  public int getPosizione() { return posizione; }

  public int getRipiano() { return ripiano; }

  public String toString() {
    return ("Posizione ("+ripiano+","+posizione+") NON valida!");
  }
}
```

e:

```
public class PosizioneNonVuota extends Exception {
  private int posizione;
  private int ripiano;

  public PosizioneNonVuota(int posizione, int ripiano) {
    this.posizione = posizione;
    this.ripiano = ripiano;
  }

  public int getPosizione() { return posizione; }

  public int getRipiano() { return ripiano; }

  public String toString() {
    return ("Posizione ("+ripiano+","+posizione+") NON vuota!");
  }
}
```

La classe *Scaffale* può quindi essere così implementata:

```
public class Scaffale {
  // attributi
  private static final int NUM_RIPIANI=5;
  private Mensola ripiani[];

  // costruttori
  public Scaffale() {
    int ripiano;

    ripiani=new Mensola[NUM_RIPIANI];
    for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++)
      ripiani[ripiano]=new Mensola();
  }

  public Scaffale(Scaffale scaffale) throws PosizioneNonValida {
    int ripiano, posizione;
    Libro libro;

    ripiani=new Mensola[NUM_RIPIANI];
    for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
      ripiani[ripiano]=new Mensola();
      for (posizione=0; posizione<ripiani[ripiano].getNumMaxVolumi();
           ripiano++) {
        libro=scaffale.getLibro(ripiano, posizione);
        if (libro!= null)
```

```
      ripiani[ripiano].setVolume(libro, posizione);
        }
      }
    }

    // metodi
    public Libro getLibro(int ripiano, int posizione)
                 throws PosizioneNonValida {
      try {
        return ripiani[ripiano].getVolume(posizione);
      }
      catch (ArrayIndexOutOfBoundsException exception) {
        throw new PosizioneNonValida(posizione, ripiano);
      }
    }

    public void setLibro(Libro libro, int ripiano, int posizione)
                throws PosizioneNonValida, PosizioneNonVuota {
      int err;
      try {err = ripiani[ripiano].setVolume(libro, posizione)
        if (err<0)
          if (err == -1)
            throw new PosizioneNonValida(posizione, ripiano);
          else
            throw new PosizioneNonVuota(posizione, ripiano);
      }
      catch (ArrayIndexOutOfBoundsException exception) {
        throw new PosizioneNonValida(posizione, ripiano);
      }
    }

    public void rimuoviLibro(int ripiano, int posizione)
                throws PosizioneNonValida {
      try {
        if (ripiani[ripiano].rimuoviVolume(posizione)<0)
          throw new PosizioneNonValida(posizione, ripiano);
      }
      catch (ArrayIndexOutOfBoundsException exception) {
        throw new PosizioneNonValida(posizione, ripiano);
      }
    }

    public int getNumRipiani() {
      return NUM_RIPIANI;
    }

    public int getNumMaxLibri() {
      int ripiano, n=0;

      for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
        n += ripiani[ripiano].getNumMaxVolumi();
      }
      return n;
    }

    public int getNumLibri() {
```

```
int ripiano, n=0;

  for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
      n += ripiani[ripiano].getNumVolumi();
    }
    return n;
  }

  public int getNumLibri(int ripiano) throws PosizioneNonValida {
    try {
      return ripiani[ripiano].getNumVolumi();
    }
    catch (ArrayIndexOutOfBoundsException exception) {
      throw new PosizioneNonValida(0, ripiano);
    }
  }
}
```

Infine il metodo *main* della classe *Scaffale* potrebbe essere riformulato come segue

```
public static void main (String[] args) {
  Scaffale scaffale = new Scaffale();
  Libro libro;

  // creazione di tre oggetti di tipo Libro
  Libro l1=new Libro("Pinocchio", "C. Collodi", 150);
  Libro l2=new Libro("Pollicino", "C. Perrault", 80);
  Libro l3=new Libro("La bella addormentata nel bosco", "C. Perrault", 50);

  // inserimento mensole #0 e #1
  try {
    scaffale.setLibro(l1, 0, 10);
    scaffale.setLibro(l2, 0, 0);
    libro = new Libro("Cappuccetto Rosso", "F.lli Grimm", 150);
    scaffale.setLibro(libro, 1, 1);
  }
  catch (PosizioneNonValida exception) {
    System.out.println(exception);
  }
  catch (PosizioneNonVuota exception) {
    System.out.println(exception);
  }

  // test errori inserimento
  try {
    scaffale.setLibro(l3, 10, 0);
  }
  catch (PosizioneNonValida exception) {
    System.out.println(exception);
  }
  catch (PosizioneNonVuota exception) {
    System.out.println(exception);
  }
  try {
    scaffale.setLibro(l3, 0, 20);
  }
  catch (PosizioneNonValida exception) {
```

```
int ripiano, n=0;

  for (ripiano=0; ripiano<NUM_RIPIANI; ripiano++) {
      n += ripiani[ripiano].getNumVolumi();
    }
    return n;
  }

  public int getNumLibri(int ripiano) throws PosizioneNonValida {
    try {
      return ripiani[ripiano].getNumVolumi();
    }
    catch (ArrayIndexOutOfBoundsException exception) {
      throw new PosizioneNonValida(0, ripiano);
    }
  }
}
```

Infine il metodo *main* della classe *Scaffale* potrebbe essere riformulato come segue

```
public static void main (String[] args) {
  Scaffale scaffale = new Scaffale();
  Libro libro;

  // creazione di tre oggetti di tipo Libro
  Libro l1=new Libro("Pinocchio", "C. Collodi", 150);
  Libro l2=new Libro("Pollicino", "C. Perrault", 80);
  Libro l3=new Libro("La bella addormentata nel bosco", "C. Perrault", 50);

  // inserimento mensole #0 e #1
  try {
    scaffale.setLibro(l1, 0, 10);
    scaffale.setLibro(l2, 0, 0);
    libro = new Libro("Cappuccetto Rosso", "F.lli Grimm", 150);
    scaffale.setLibro(libro, 1, 1);
  }
  catch (PosizioneNonValida exception) {
    System.out.println(exception);
  }
  catch (PosizioneNonVuota exception) {
    System.out.println(exception);
  }

  // test errori inserimento
  try {
    scaffale.setLibro(l3, 10, 0);
  }
  catch (PosizioneNonValida exception) {
    System.out.println(exception);
  }
  catch (PosizioneNonVuota exception) {
    System.out.println(exception);
  }
  try {
    scaffale.setLibro(l3, 0, 20);
  }
  catch (PosizioneNonValida exception) {
```

```
        System.out.println(exception);
    }
    catch (PosizioneNonVuota exception) {
        System.out.println(exception);
    }
    try {
        scaffale.setLibro(13, 0, 10);
    }
    catch (PosizioneNonValida exception) {
        System.out.println(exception);
    }
    catch (PosizioneNonVuota exception) {
        System.out.println(exception);
    }

    // inserimento mensola #1
    try {
        scaffale.setLibro(13, 1, 0);
    }
    catch (PosizioneNonValida exception) {
        System.out.println(exception);
    }
    catch (PosizioneNonVuota exception) {
        System.out.println(exception);
    }

    // visualizzazione contenuto mensole
    try {
        for (int ripiano=0; ripiano<scaffale.getNumRipiani(); ripiano++) {
            for (int posizione=0; posizione<scaffale.getNumLibri(ripiano);
                posizione++) {
                libro = scaffale.getLibro(ripiano, posizione);
                if (libro!= null)
                    System.out.println("ripiano: "+ripiano+ " posizione: "+posizione+" -> "+
                                        libro.getTitolo()+" "+libro.prezzo()+"€");
            }
        }
    }
    catch (PosizioneNonValida exception) {
        System.out.println(exception);
    }

    // modifica titolo e autore libro estratto da scaffale
    try {
        libro=scaffale.getLibro(0,0);
        if (libro!= null) {
            libro.setTitolo("Sussi e Biribissi");
            libro.setAutore("Collodi nipote");
        }
    }
    catch (PosizioneNonValida exception) {
        System.out.println(exception);
    }

    // visualizzazione contenuto mensole
    try {
```

```
    System.out.println(exception);
  }
  catch (PosizioneNonVuota exception) {
    System.out.println(exception);
  }
  try {
    scaffale.setLibro(13, 0, 10);
  }
  catch (PosizioneNonValida exception) {
    System.out.println(exception);
  }
  catch (PosizioneNonVuota exception) {
    System.out.println(exception);
  }

  // inserimento mensola #1
  try {
    scaffale.setLibro(13, 1, 0);
  }
  catch (PosizioneNonValida exception) {
    System.out.println(exception);
  }
  catch (PosizioneNonVuota exception) {
    System.out.println(exception);
  }

  // visualizzazione contenuto mensole
  try {
    for (int ripiano=0; ripiano<scaffale.getNumRipiani(); ripiano++) {
      for (int posizione=0; posizione<scaffale.getNumLibri(ripiano);
           posizione++) {
        libro = scaffale.getLibro(ripiano, posizione);
        if (libro!= null)
          System.out.println("ripiano: "+ripiano+ " posizione: "+posizione+" -> "+
                             libro.getTitolo()+" "+libro.prezzo()+"€");
      }
    }
  }
  catch (PosizioneNonValida exception) {
    System.out.println(exception);
  }

  // modifica titolo e autore libro estratto da scaffale
  try {
    libro=scaffale.getLibro(0,0);
    if (libro!= null) {
        libro.setTitolo("Sussi e Biribissi");
        libro.setAutore("Collodi nipote");
    }
  }
  catch (PosizioneNonValida exception) {
    System.out.println(exception);
  }

  // visualizzazione contenuto mensole
  try {
```

```
    for (int ripiano=0; ripiano<scaffale.getNumRipiani(); ripiano++) {
      for (int posizione=0; posizione<scaffale.getNumLibri(ripiano);
           posizione++) {
        libro = scaffale.getLibro(ripiano, posizione);
        if (libro!= null)
          System.out.println("ripiano: "+ripiano+ " posizione: "+posizione+" -> "+
                             libro.getTitolo()+" "+libro.prezzo()+"€");
        }
      }
    }
    catch (PosizioneNonValida exception) {
      System.out.println(exception);
    }
  }
```

e produce il seguente output:

```
Posizione (10,0) NON valida!
Posizione (0,20) NON valida!
Posizione (0,10) NON vuota!
ripiano: 0 posizione: 0 -> Pollicino 9.5€
ripiano: 1 posizione: 0 -> La bella addormentata nel bosco 8.0€
ripiano: 1 posizione: 1 -> Cappuccetto Rosso 13.0€
ripiano: 0 posizione: 0 -> Pollicino 9.5€
ripiano: 1 posizione: 0 -> La bella addormentata nel bosco 8.0€
ripiano: 1 posizione: 1 -> Cappuccetto Rosso 13.0€
```

OSSERVAZIONE Come si può vedere dall'implementazione del metodo *main* dell'esempio precedente, se il codice contenuto in un blocco `try` genera potenzialmente più di un'eccezione, è possibile associare al blocco più blocchi `catch`, uno per ciascun tipo di eccezione.

5.1 *Catching* multiplo di tipi di eccezione e propagazione del sollevamento di eccezioni con controllo di tipo

Catching multiplo di tipi di eccezione

A partire dalla versione 7 di Java, con un singolo blocco `catch` è possibile gestire più tipi di eccezioni così da eliminare ridondanze di codice.

Per esempio, una situazione come quella che segue dove la variabile *ex* ha tipi diversi e che nelle versioni precedenti di Java viene codificata con

```
catch (IOException ex) {
  codiceDaEseguire();
  throw ex;
}
catch (SQLException ex) {
  codiceDaEseguire();
  throw ex;
}
```

```
    for (int ripiano=0; ripiano<scaffale.getNumRipiani(); ripiano++) {
      for (int posizione=0; posizione<scaffale.getNumLibri(ripiano);
           posizione++) {
        libro = scaffale.getLibro(ripiano, posizione);
        if (libro!= null)
          System.out.println("ripiano: "+ripiano+ " posizione: "+posizione+" -> "+
                             libro.getTitolo()+" "+libro.prezzo()+"€");
        }
      }
    }
    catch (PosizioneNonValida exception) {
      System.out.println(exception);
    }
  }
```

e produce il seguente output:

```
Posizione (10,0) NON valida!
Posizione (0,20) NON valida!
Posizione (0,10) NON vuota!
ripiano: 0 posizione: 0 -> Pollicino 9.5€
ripiano: 1 posizione: 0 -> La bella addormentata nel bosco 8.0€
ripiano: 1 posizione: 1 -> Cappuccetto Rosso 13.0€
ripiano: 0 posizione: 0 -> Pollicino 9.5€
ripiano: 1 posizione: 0 -> La bella addormentata nel bosco 8.0€
ripiano: 1 posizione: 1 -> Cappuccetto Rosso 13.0€
```

OSSERVAZIONE Come si può vedere dall'implementazione del metodo *main* dell'esempio precedente, se il codice contenuto in un blocco `try` genera potenzialmente più di un'eccezione, è possibile associare al blocco più blocchi `catch`, uno per ciascun tipo di eccezione.

5.1 *Catching* multiplo di tipi di eccezione e propagazione del sollevamento di eccezioni con controllo di tipo

Catching multiplo di tipi di eccezione

A partire dalla versione 7 di Java, con un singolo blocco `catch` è possibile gestire più tipi di eccezioni così da eliminare ridondanze di codice.

Per esempio, una situazione come quella che segue dove la variabile *ex* ha tipi diversi e che nelle versioni precedenti di Java viene codificata con

```
catch (IOException ex) {
  codiceDaEseguire();
  throw ex;
}
catch (SQLException ex) {
  codiceDaEseguire();
  throw ex;
}
```

in Java 7 e successive, diventa:

```
catch (IOException|SQLException ex) {
  codiceDaEseguire();
  throw ex;
}
```

Nell'istruzione `catch` i vari tipi di eccezioni che il relativo blocco può gestire vengono dichiarati separati dalla barra verticale «|». Se un blocco `catch` gestisce più di un tipo di eccezione il parametro del `catch` (nel nostro esempio *ex*) è implicitamente costante (*final*) e quindi immutabile nell'ambito del blocco stesso.

Il *byte-code* generato compilando un blocco `catch` che gestisce eccezioni multiple risulta più compatto (e quindi migliore) rispetto a quello della compilazione di più blocchi `catch` che gestiscono ognuno uno specifico tipo di eccezione.

Propagazione del sollevamento di eccezioni con controllo di tipo

Rispetto al meccanismo di propagazione del sollevamento delle eccezioni, i compilatori Java 7 e successivi sono in grado di gestire la specifica di più tipi di eccezioni nella clausola `throws` nella dichiarazione di un metodo.

Nell'esempio che segue il blocco `try` può lanciare *EccezioneA* o *EccezioneB* e l'eventuale propagazione dell'eccezione *ex* (blocco `catch`) è corretta perché essendo *ex* di tipo *Exception*, è anche un loro supertipo:

```
  static class EccezioneA extends Exception { }
  static class EccezioneB extends Exception { }

  public void propagaEccezione(String nomeEccezione)
              throws Exception {
    try {
      if (nomeEccezione.equals("A")) {
        throw new EccezioneA();
      } else {
        throw new EccezioneB();
      }
    } catch (Exception ex) {
      throw ex;
    }
  }
```

Supponiamo ora di specificare *EccezioneA* o *EccezioneB* nella clausola `throws` della dichiarazione del metodo *propagaEccezione* invece di *Exception*:

- nelle versioni precedenti Java 7 questo non è possibile perché essendo il parametro *ex* della clausola `catch` di tipo *Exception* tale blocco può rilanciare solo eccezioni di tipo *Exception*;
- a partire dalla versione 7, specificando più tipi di eccezione nella clausola `throws` di un metodo, in caso di propagazione di un'eccezione il compilatore è in grado di riconoscerne il tipo. Nel nostro esempio, sa-

in Java 7 e successive, diventa:

```
catch (IOException|SQLException ex) {
  codiceDaEseguire();
  throw ex;
}
```

Nell'istruzione `catch` i vari tipi di eccezioni che il relativo blocco può gestire vengono dichiarati separati dalla barra verticale «|». Se un blocco `catch` gestisce più di un tipo di eccezione il parametro del `catch` (nel nostro esempio *ex*) è implicitamente costante (*final*) e quindi immutabile nell'ambito del blocco stesso.

Il *byte-code* generato compilando un blocco `catch` che gestisce eccezioni multiple risulta più compatto (e quindi migliore) rispetto a quello della compilazione di più blocchi `catch` che gestiscono ognuno uno specifico tipo di eccezione.

Propagazione del sollevamento di eccezioni con controllo di tipo

Rispetto al meccanismo di propagazione del sollevamento delle eccezioni, i compilatori Java 7 e successivi sono in grado di gestire la specifica di più tipi di eccezioni nella clausola `throws` nella dichiarazione di un metodo.

Nell'esempio che segue il blocco `try` può lanciare *EccezioneA* o *EccezioneB* e l'eventuale propagazione dell'eccezione *ex* (blocco `catch`) è corretta perché essendo *ex* di tipo *Exception*, è anche un loro supertipo:

```
  static class EccezioneA extends Exception { }
  static class EccezioneB extends Exception { }

  public void propagaEccezione(String nomeEccezione)
              throws Exception {
    try {
      if (nomeEccezione.equals("A")) {
        throw new EccezioneA();
      } else {
        throw new EccezioneB();
      }
    } catch (Exception ex) {
      throw ex;
    }
  }
```

Supponiamo ora di specificare *EccezioneA* o *EccezioneB* nella clausola `throws` della dichiarazione del metodo *propagaEccezione* invece di *Exception*:

- nelle versioni precedenti Java 7 questo non è possibile perché essendo il parametro *ex* della clausola `catch` di tipo *Exception* tale blocco può rilanciare solo eccezioni di tipo *Exception*;
- a partire dalla versione 7, specificando più tipi di eccezione nella clausola `throws` di un metodo, in caso di propagazione di un'eccezione il compilatore è in grado di riconoscerne il tipo. Nel nostro esempio, sa-

pendo che l'eccezione *ex* lanciata da un'istruzione `throw` del blocco `try` può essere solo di tipo *EccezioneA* o *EccezioneB*, anche se *ex* è di tipo *Exception*, il compilatore è in grado di inferire la sua corretta classe di appartenenza.

In generale, in Java 7 e successive, quando si dichiara uno o più tipi di eccezione in una clausola `catch` e si propaga l'eccezione gestita nel relativo blocco `catch` il compilatore controlla che il tipo di eccezione propagato verifichi le seguenti condizioni:

- il blocco `try` sia abilitato a propagarlo;
- non esistano altri precedenti blocchi `catch` che possono gestirlo;
- sia un sottotipo o supertipo di uno dei parametri della clausola `catch`.

6 Gestione dell'input/output predefinito

Nel corso degli esempi presentati si sono utilizzate più volte istruzioni del tipo

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

che visualizzano i risultati di un programma sullo standard output di sistema (normalmente una finestra testuale sullo schermo). *System.out* è un oggetto predefinito di classe *PrintStream* associato allo standard output, così come *System.err* normalmente utilizzato per la visualizzazione dei messaggi di errore. L'oggetto *System.out* dispone di metodi – come *println* – che accettano come parametro una stringa, o un tipo di dato primitivo da visualizzare. Il metodo *print* – a differenza del metodo *println* – non effettua il ritorno a capo dopo aver visualizzato il parametro passato come argomento.

OSSERVAZIONE Entrambi i metodi *println* e *print* accettano un unico argomento, ma utilizzando l'operatore di concatenazione tra stringhe «+», che ha la proprietà di convertire eventuali valori di tipi di dato predefiniti, o di oggetti di classi che implementano il metodo standard *toString*, è possibile effettuare visualizzazioni composite.

Per le operazioni di input esiste un oggetto predefinito analogo associato allo standard input (normalmente la tastiera): *System.in* di classe *InputStream*. La maggiore complessità delle operazioni di input rende necessario clonare l'oggetto e incapsularlo in un oggetto di classe *BufferedReader* che ne semplifica le funzionalità:

```
InputStreamReader input = new InputStreamReader(System.in);
BufferedReader keyboard = new BufferedReader(input);
```

In questo modo viene definito un oggetto *keyboard* di classe *BufferedReader* che rende disponibile il metodo *readLine*, che consente di leggere dallo standard input un'intera riga di testo.

ESEMPIO

Le seguenti istruzioni effettuano la lettura di una stringa di caratteri digitata sulla tastiera:

```
String s;
s = keyboard.readLine();
```

L'operazione deve essere effettuata gestendo le eccezioni di classe *IOException* (del *package java.io*) che possono essere generate dal metodo *readLine*:

```
String s;
try {
  s = keyboard.readLine();
}
catch (java.io.IOException exception) {
  ...
  ...
  ...
}
```

Nel caso in cui si debbano leggere dei valori numerici, è necessario convertire la stringa comunque restituita dal metodo *readLine* in un valore numerico.

I metodi della seguente classe di utilità *ConsoleInput* consentono di acquisire da tastiera in modo controllato vari tipi di dato:

```
import java.io.*;

/**
 * classe per l'acquisizione di stringhe e numeri dallo
 * standard input
 */
public class ConsoleInput {
  BufferedReader reader;

  /**
   * costruisce un oggetto di tipo BufferedReader sopra lo
   * standard input (System.in)
   */
  public ConsoleInput() {
    reader = new BufferedReader(new InputStreamReader
                                (System.in));
  }

  /**
   * legge una riga in input e la converte in un valore
   * intero: la linea deve contenere un solo valore intero
   * eventualmente preceduto dal segno, ma senza altri
   * caratteri.
   * @return valore intero digitato
   */
  public int readInt() throws IOException {
    return Integer.parseInt(reader.readLine());
  }

  /**
   * legge una riga in input e la converte in un valore
```

```
  * floating-point: la linea deve contenere un solo valore
  * decimale eventualmente preceduto dal segno, ma senza
  * altri caratteri.
  * @return valore decimale digitato
  */
  public double readDouble() throws IOException {
    return Double.parseDouble(reader.readLine());
  }

  /**
   * legge una riga di testo in input.
   * @return stringa digitata
   */
  public String readLine() throws IOException {
    return reader.readLine();
  }
}
```

OSSERVAZIONE I metodi statici di conversione delle classi *Integer* (*parseInt*) e *Double* (*parseDouble*) generano un'eccezione di tipo *NumberFormatException* che, essendo sottoclasse di *RuntimeException*, è di tipo *unchecked* e quindi non deve necessariamente essere intercettata, o dichiarata, lasciandone l'eventuale gestione al codice chiamante.

ESEMPIO La classe *InputCoin* ha un unico metodo *main* che esemplifica l'uso delle funzionalità rese disponibili dalla classe *ConsoleInput*:

```
public class InputCoin {
  public static void main(String[] args) {
    int centesimi = 0;
    double euro = 0.0, totale;
    boolean ok;
    String chi = "";

    // crea un oggetto ConsoleInput
    ConsoleInput keyboard = new ConsoleInput();

    System.out.println("Quanti centesimi hai?");
    ok = false;
    while(!ok) {
      try {
            centesimi = keyboard.readInt();
            ok = true;
          }
      catch(java.io.IOException exception) {
        System.out.println("Valore non corretto. Reinseriscilo!");
      }
      catch(NumberFormatException exception) {
        System.out.println("Valore non corretto. Reinseriscilo!");
      }
    }

    System.out.println("Supponiamo di avere 75 centesimi "+
                       "di Euro, quanti Euro sono?");
```

```
      ok=false;
      while(!ok) {
        try {
              euro = keyboard.readDouble();
              ok = true;
            }
        catch(java.io.IOException exception) {
          System.out.println("Valore non corretto. Reinseriscilo!");
        }
        catch(NumberFormatException exception) {
          System.out.println("Valore non corretto. Reinseriscilo!");
        }
        if (ok)
          if (euro != 0.75) {
            System.out.println("I conti non tornano! Quanti Euro "+
                               "sono 75 centesimi?");
            ok=false;
          }
        }

      System.out.println("A chi devi questi soldi?");
      try {
            chi = keyboard.readLine();
      }
      catch(java.io.IOException exception) {
      }

      totale = centesimi*0.01 + euro;
      System.out.println("Devi "+totale+"€ a "+chi);
    }
  }
```

OSSERVAZIONE Nell'esempio precedente l'invocazione del metodo *readLine* dell'oggetto *keyboard* istanza della classe *ConsoleInput* deve necessariamente essere inserita in un blocco `try/catch`, perché è dichiarata la possibile generazione dell'eccezione di tipo *IOException*; ma, dato che in pratica questa eccezione non può essere sollevata da nessun possibile input da parte dell'utente, il blocco `catch` è stato lasciato vuoto.

7 Gestione dell'input/output da file di testo

Con tecniche di programmazione simili a quelle utilizzate nel paragrafo precedente per la gestione dell'input/output predefinito, è possibile gestire l'input/output da/in un file di testo. I metodi della seguente classe di utilità *TextFile* consentono di aprire, in lettura o scrittura, e chiudere un file di testo e di leggere o scrivere da/in esso stringhe corrispondenti a linee di testo:

```
import java.io.*;

/**
 * classe per la gestione sequenziale di un file di testo      ▶
```

```
*/
public class TextFile {
  private char mode; // R=read, W=write
  private BufferedReader reader;
  private BufferedWriter writer;

/**
 * costruisce un oggetto di tipo BufferedReader/
 * BufferedWriter
 * sopra il file specificato dal nome indicato.
 * @param filename percorso e nome del file
 * @param mode R per sola lettura, W per sola scrittura
 */
public TextFile(String filename, char mode) throws IOException
{
  this.mode = 'R';
  if (mode == 'W' || mode == 'w') {
    this.mode = 'W';
    writer = new BufferedWriter(new FileWriter(filename));
  }
  else {
    reader = new BufferedReader(new FileReader(filename));
  }
}

/**
 * scrive una riga di testo in un file aperto in scrittura.
 * @param line riga di testo da scrivere nel file
 * @throws FileException se il file è aperto in lettura
 */
public void toFile(String line) throws FileException,
                                IOException
{
  if (this.mode == 'R') throw new FileException("Read-only"+
                                                "file!");
  writer.write(line);
  writer.newLine();
}

/**
 * legge una riga di testo da un file aperto in lettura.
 * @return riga di testo letta dal file
 * @throws FileException se il file è aperto in scrittura
 * @throws FileException se è stata raggiunta la fine
 * del file
 */
public String fromFile() throws FileException, IOException
{
  String tmp;

if (this.mode == 'W') throw new FileException("Write-only "+
                                              "file!");
  tmp = reader.readLine();
```

```
    if (tmp == null) throw new FileException("End of file!");
    return tmp;
  }

  /**
   * chiude il file aperto dal costruttore
   */
  public void closeFile() throws IOException
  {
    if (this.mode == 'W')
      writer.close();
    else // this.mode == 'R'
      reader.close();
  }

  public static void main(String args[]) throws IOException
  {
    TextFile out = new TextFile("file.txt",'W');
    try {
          out.toFile("Riga 1");
          out.toFile("Riga 2");
          out.toFile("Riga 3");
    }
    catch (FileException exception) {
      System.out.println(exception.getMatter());
    }
    out.closeFile();

    TextFile in = new TextFile("file.txt",'R');
    String line;
    try {
          while (true) {
            line = in.fromFile();
            System.out.println(line);
          }
    }
    catch (FileException exception) {
      System.out.println(exception.getMatter());
    }
    out.closeFile();
  }
}
```

La classe di utilità *TextFile* genera eccezioni specifiche di tipo *FileException* definite dalla seguente classe che consente, nel costruttore, di specificare un messaggio descrittivo dell'errore che ha causato l'anomalia:

```
public class FileException extends Exception {
  private String matter="";

  public FileException(String matter) {
    this.matter = matter;
  }
  public String getMatter() {
```

```
    return this.matter;
  }
}
```

OSSERVAZIONE Il metodo *main* della classe *TextFile*, oltre a costituire un test dei metodi della classe, rappresenta un esempio di uso della stessa. In particolare, il metodo *fromFile* genera un'eccezione di tipo *FileException* quando viene invocato dopo che è stata raggiunta la fine del file per cui un normale ciclo di lettura di tutte le righe è un ciclo potenzialmente infinito da cui si esce intercettando l'eccezione. L'esecuzione del metodo *main*, oltre a creare un file denominato *file.txt* costituito da 3 righe di testo – rispettivamente «Riga 1», «Riga 2» e «Riga 3» –, produce il seguente output

```
Riga 1
Riga 2
Riga 3
End of file!
```

dove l'ultima riga è prodotta dalla visualizzazione del messaggio dell'eccezione sollevata dall'invocazione del metodo *fromFile* dopo la lettura dell'ultima riga del file.

ESEMPIO Volendo dotare la classe *Mensola* della possibilità di salvare/ripristinare su/da file l'elenco dei libri che essa contiene, è possibile aggiungere i seguenti metodi:

```
public void salvaVolumi() throws java.io.IOException {
  TextFile out = new TextFile("volumi.txt", 'W');

  try {
        for (int posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
          if (volumi[posizione]!= null) {
            String line = Integer.toString(posizione);
            line += ";"+volumi[posizione].getTitolo();
            line += ";"+volumi[posizione].getAutore();
            line += ";"+volumi[posizione].getNumeroPagine();
            out.toFile(line);
          }
      }
  catch (FileException exception) {
  }
  out.closeFile();
}

public void caricaVolumi() throws java.io.IOException {
  TextFile in = new TextFile("volumi.txt", 'R');
  int posizione, pagine;
  String linea, autore, titolo;
  String[] elementi;
  Libro libro;

  try {
        while(true) {
          linea = in.fromFile();
          elementi = linea.split(";");
```

```
            if (elementi.length == 4) {
              posizione = Integer.parseInt(elementi[0]);
              titolo = elementi[1];
              autore = elementi[2];
              pagine = Integer.parseInt(elementi[3]);
              libro = new Libro(titolo, autore, pagine);
              setVolume(libro, posizione);
            }
        }
      }
  catch (FileException exception) {
  }
}
```

Il metodo *salvaVolumi* crea un file di testo in formato CSV (*Comma Separated Values*) in cui ogni riga corrisponde a un volume e i valori della posizione, del titolo, dell'autore e del numero di pagine sono separati dal carattere «;», come nel seguente esempio:

```
0;Pinocchio;C. Collodi;150
2;Pollicino;C. Perrault;80
10;La bella addormentata nel bosco;C. Perrault;50
```

Il metodo *caricaVolumi* legge ogni singola riga del file di testo e utilizza il metodo *split* della classe *String* per separare in un *array* di stringhe (il vettore *elementi*) gli elementi della riga del file separati dal carattere fornito come parametro (in questo caso il carattere «;»). A partire da queste sottostringhe della riga del file, che in alcuni casi devono essere convertite in valori numerici, viene costruito un oggetto di classe *Libro* che viene fornito come argomento al metodo *setVolume* della classe *Mensola*, per inserire il volume caricato dal file nella posizione indicata. Il seguente metodo *main* serve a testare i metodi illustrati

```
public static void main (String[] args) throws java.io.IOException {
  Mensola mensola = new Mensola();

  Libro l1 = new Libro("Pinocchio", "C. Collodi", 150);
  Libro l2 = new Libro("Pollicino", "C. Perrault", 80);
  Libro l3 = new Libro("La bella addormentata nel bosco", "C. Perrault", 50);
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // inserimento volumi
  mensola.setVolume(l1, 0);
  mensola.setVolume(l2, 2);
  mensola.setVolume(l3, 10);
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // salvataggio volumi su file
  mensola.salvaVolumi();
  // creazione nuova mensola vuota
  mensola = new Mensola();
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // caricamento volumi da file
  mensola.caricaVolumi();
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
}
```

e produce il seguente output

```
Numero volumi: 0
Numero volumi: 3
Numero volumi: 0
Numero volumi: 3
```

che dimostra il successo del caricamento dei volumi dal file dove erano stati in precedenza salvati.

8 Serializzazione e persistenza degli oggetti su file

La **serializzazione** di un oggetto è un processo che permette di salvarlo in un supporto di memorizzazione sequenziale (come un file), o di trasmetterlo su un canale di comunicazione sequenziale (come una connessione di rete). La serializzazione può essere effettuata in forma binaria, oppure può impiegare codifiche testuali come il formato XML (*eXtensible Markup Language*). Lo scopo della serializzazione è quello di salvare e/o trasmettere l'intero stato dell'oggetto in modo che esso possa essere successivamente ricreato nello stesso identico stato dal processo inverso, denominato **deserializzazione**.

ESEMPIO

Volendo salvare lo stato di un oggetto istanza della classe *Mensola* introdotta in precedenza, possiamo aggiungere il seguente metodo che serializza e memorizza in un file denominato *volumi.bin* l'*array volumi* unico attributo della classe:

```
public void salvaMensola() throws java.io.IOException {
  ObjectOutputStream stream =
    new ObjectOutputStream(new FileOutputStream("volumi.bin"));

  stream.writeObject(this.volumi);
  stream.close();
}
```

La deserializzazione dell'*array volumi* può essere effettuata a partire dal contenuto del file mediante il seguente metodo:

```
public void caricaMensola() throws java.io.IOException {
  ObjectInputStream stream =
    new ObjectInputStream(new FileInputStream("volumi.bin"));

  try {
        this.volumi = (Libro[])stream.readObject();
  }
  catch (ClassNotFoundException exception) {
  }
  stream.close();
}
```

È necessario in questo caso effettuare un *casting* di quanto restituito dal metodo *readObject* al tipo dell'attributo cui lo si assegna (in questo caso un *array* di oggetti di classe *Libro*). Questa operazione, in caso di discordanza tra il tipo specificato e quanto contenuto nel file, può generare un'eccezione di tipo *ClassNotFoundException* che deve essere di conseguenza intercettata.

OSSERVAZIONE Nel caso che una classe abbia più attributi è possibile serializzarli all'interno dello stesso file; è però necessario che l'ordine di invocazione dei metodi *readObject* sia rigorosamente lo stesso con cui sono stati invocati i metodi *writeObject*, per evitare di assegnare un oggetto a un riferimento incoerente, generando un'eccezione di tipo *ClassNotFoundException*.

Le classi degli oggetti che devono essere serializzati devono implementare l'interfaccia *Serializable* del *package java.io*: se hanno come attributi istanze di altre classi, ciascuna di esse deve implementare la stessa interfaccia. Dato che *Serializable* è un'interfaccia vuota, non è necessario definire alcun metodo particolare.

Nel caso specifico della classe *Libro*, cui si riferisce l'esempio precedente, è sufficiente ridefinirne la dichiarazione nel seguente modo:

```
public class Libro implements java.io.Serializable {
  ...
  ...
  ...
}
```

ESEMPIO

Per testare la funzionalità del codice visto è possibile modificare il metodo *main* della classe *Mensola* come segue:

```
public static void main (String[] args) throws java.io.IOException {
  Mensola mensola = new Mensola();

  Libro l1 = new Libro("Pinocchio", "C. Collodi", 150);
  Libro l2 = new Libro("Pollicino", "C. Perrault", 80);
  Libro l3 = new Libro("La bella addormentata nel bosco", "C. Perrault", 50);
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // inserimento volumi
  mensola.setVolume(l1, 0);
  mensola.setVolume(l2, 2);
  mensola.setVolume(l3, 10);
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // visualizzazione elenco volumi
  for (int posizione=0; posizione<mensola.getNumMaxVolumi(); posizione++) {
    Libro libro = mensola.getVolume(posizione);
    if (libro != null)
      System.out.println("posizione: "+posizione+" -> "+libro.getTitolo()+" "+
                         libro.prezzo()+"€");
  }
  // salvataggio volumi su file
  mensola.salvaMensola();
  // creazione nuova mensola vuota
  mensola = new Mensola();
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // caricamento volumi da file
  mensola.caricaMensola();
  System.out.println("Numero volumi: "+mensola.getNumVolumi());
  // visualizzazione elenco volumi
  for (int posizione=0; posizione<mensola.getNumMaxVolumi();
       posizione++) {
    Libro libro = mensola.getVolume(posizione);
    if (libro != null)
      System.out.println("posizione: "+posizione+" -> "+libro.getTitolo()+" "+
                         libro.prezzo()+"€");
  }
}
```

L'output prodotto dimostra il successo del caricamento del vettore *volumi* dal file dove era stato in precedenza salvato:

```
Numero volumi: 0
Numero volumi: 3
posizione: 0 -> Pinocchio 13.0€
posizione: 2 -> Pollicino 9.5€
posizione: 10 -> La bella addormentata nel bosco 8.0€
Numero volumi: 0
Numero volumi: 3
posizione: 0 -> Pinocchio 13.0€
posizione: 2 -> Pollicino 9.5€
posizione: 10 -> La bella addormentata nel bosco 8.0€
```

La serializzazione in un file degli attributi di un oggetto consente di renderlo **persistente**: è infatti sufficiente invocare il metodo che serializza gli attributi su file prima della sua distruzione e richiamare il metodo che deserializza gli attributi da file nel costruttore (intercettando l'eventuale eccezione generata nel caso che il file non esista), per fare in modo che un oggetto sopravviva all'esecuzione del programma in cui viene creato.

I file in cui sono serializzati gli oggetti Java hanno un formato binario, per cui non è possibile ispezionarli utilizzando un normale editor. La serializzazione di un oggetto può esporre i dettagli implementativi della classe di cui è istanza; spesso i produttori software non documentano il formato di serializzazione e in alcuni casi cifrano i dati serializzati.

I CONCETTI CHIAVE

ARRAY. Gli *array* in Java sono oggetti: è possibile sia definire *array* i cui elementi contengono tipi di dato primitivi, sia *array* i cui elementi contengono riferimenti a oggetti di una determinata classe. Per istanziare un *array* è possibile utilizzare un'istruzione come la seguente

tipo identificatore[ ] = **new** *tipo*[*dimensione*] ;

per esempio

```
String vettoreDiStringhe[] = new String[10];
```

Nel caso di un *array* di riferimenti a oggetti, un elemento che non riferisce alcun oggetto contiene il valore **null**. Dal momento che un *array* può contenere oggetti e che un *array* è esso stesso un oggetto, è possibile creare strutture comunque complesse. L'attributo *length* di un *array* contiene il numero dei suoi elementi. In Java è possibile sia passare *array* come parametri ai metodi sia ricevere da questi *array* come valori di ritorno.

COSTRUTTORE DI COPIA. Se gli attributi di una classe non sono di un tipo di dato primitivo o immutabile (come per esempio gli oggetti di classe *String*), ma riferimenti a oggetti, è necessario che il codice del costruttore di copia li cloni senza limitarsi a copiarne i riferimenti; in caso contrario il costruttore crea un nuovo oggetto che condivide con l'oggetto originale i riferimenti agli attributi, e ogni variazione dell'uno si rifletterebbe in una automatica e spesso inaspettata variazione dell'altro. La stessa attenzione deve essere posta ogni volta che un metodo di una classe restituisce un riferimento a un attributo, o ha come parametro un oggetto da riferire da parte di un attributo: è sempre necessario effettuare una clonazione dell'oggetto.

ECCEZIONI. Sono eventi che si presentano in fase di esecuzione (*run-time*) di un programma al verificarsi di situazioni anomale (divisione per zero, uso di un indice fuori dal *range* di un *array*, accesso a un riferimento nullo, ...). Un'eccezione può essere intercettata e gestita, oppure il programma termina la sua esecuzione. L'intercettazione e la gestione delle eccezioni si basano sul costrutto **try/catch**. La parola chiave **try** permette di definire un blocco di istruzioni di cui controllare l'esecuzione per intercettare eventuali eccezioni che potrebbero verificarsi; ogni singola clausola **catch** definisce un blocco di istruzioni che devono recuperare la situazione anomala connessa alla specifica eccezione intercettata. In Java le eccezioni sono oggetti istanza di classi che derivano dalla classe predefinita *Exception*, che a sua volta deriva da *Throwable*; il linguaggio definisce alcune classi di eccezioni predefinite: *ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException, IOException*, ... Una classe eccezione può essere minimale come la seguente

```
public class Eccezione extends Exception {};
```

oppure il programmatore può definire attributi, costruttori e metodi per le necessità di gestione degli oggetti che costituiscono le eccezioni concretamente sollevate, in particolare per fornire informazioni specifiche sul tipo di errore che ha causato l'eccezione stessa.

GENERAZIONE DI ECCEZIONI. Per segnalare il proprio insuccesso, un metodo di una classe Java può generare un'eccezione. Un metodo che può generare un'eccezione deve specificare questa eventualità nella propria firma utilizzando la parola chiave **throws** seguita dal tipo di eccezione, o da un elenco di tipi di eccezioni, che possono essere sollevate dal metodo. Nel codice del metodo la parola chiave **throw** consente di interrompere l'esecuzione e di sollevare l'eccezione specificata come un nuovo oggetto. La semantica dell'istruzione **throw** ha alcuni aspetti in comune con quella dell'istruzione **return**: in particolare l'esecuzione di **throw** implica la terminazione immediata del metodo e il passaggio del controllo al codice chiamante del metodo stesso. Un metodo può sollevare più tipi di eccezione, che possono essere singolarmente gestite specificando clausole **catch** multiple.

HANDLE OR DECLARE. Questa regola («gestisci o dichiara») stabilisce che, a fronte di una possibile eccezione di tipo controllato (*checked*), il codice di un metodo deve intercettarla e gestirla, oppure dichiarare a sua volta di sollevarla; un'eccezione non deve mai passare inosservata: se un metodo non la gestisce, trasferisce l'obbligo di gestirla al metodo che lo ha invocato. Alcune eccezioni predefinite di classe *RuntimeException* sono di tipo non controllato (*unchecked*) e possono essere non gestite e non dichiarate: in questo caso la loro eventuale generazione causa l'interruzione dell'esecuzione del programma.

INPUT/OUTPUT PREDEFINITO. Gli oggetti predefiniti *System.out* di classe *PrintStream* e *System.in* di classe *InputStream* sono normalmente associati rispettivamente allo standard output (finestra di tipo testuale sullo schermo) e standard input (tastiera). I metodi che rendono disponibili consentono di eseguire operazioni di input/output tradizionali (per esempio utilizzando i metodi *print* o *println* della classe *PrintStream*): la complessità della classe *InputStream* può essere gestita incapsulandolo in un oggetto di tipo *BufferedReader* che rende disponibile il metodo *readLine* per la lettura di una riga di testo dallo standard input.

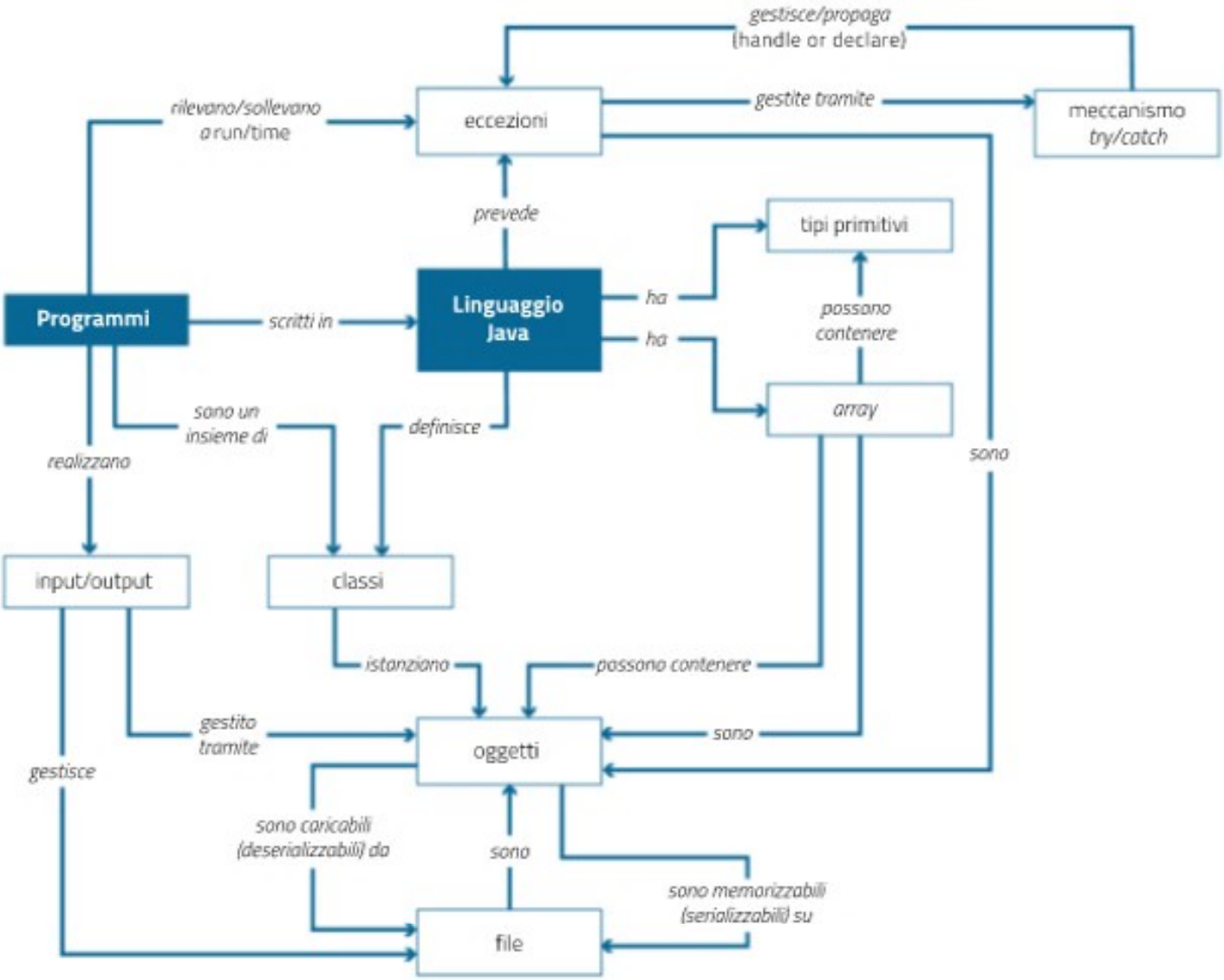
INPUT/OUTPUT SEQUENZIALE DA FILE DI TESTO. Le classi *BufferedReader* e *BufferedWriter*, che incapsulano a loro volta oggetti di tipo *FileReader* e *FileWriter*, rendono disponibili semplici metodi per la lettura/scrittura di singole righe di un file di testo.

SERIALIZZAZIONE E PERSISTENZA DEGLI OGGETTI. La serializzazione di un oggetto è un processo che permette di salvarlo in un supporto di memorizzazione sequenziale (come un file), o di trasmetterlo su un canale di comunicazione sequenziale (come una connessione di rete). La serializzazione può essere effettuata in forma binaria, oppure può impiegare codifiche testuali come il formato XML (*eXtensible Markup Language*). Lo scopo della serializzazione è quello di salvare e/o trasmettere l'intero stato dell'oggetto in modo che esso possa essere successivamente ricreato nello stesso identico stato dal processo inverso, denominato *deserializzazione*. Una classe i cui oggetti devono essere serializzati deve implementare l'interfaccia *Serializable* del *package java.io*:

```
public class ClasseSerializzabile
      implements java.io.Serializable {
    ...
    ...
    ...
}
```

Se tale classe ha come attributi dei tipi di dato non primitivi, anche le classi di cui sono istanza devono a loro volta implementare l'interfaccia *Serializable*. Un oggetto serializzabile può essere reso persistente su file invocando il metodo *writeObject* della classe *ObjectOutputStream*, che incapsula un oggetto di tipo *FileOutputStream*. L'oggetto può essere ripristinato dal file invocando il metodo *readObject* della classe *ObjectInputStream*, che incapsula un oggetto di tipo *FileInputStream*.

RIPASSA CON LA MAPPA



SPIEGA IN UNA PAGINA

1 Descrivere in che cosa consiste il meccanismo delle eccezioni.

2 Descrivere il meccanismo della serializzazione e quali sono i suoi scopi.

QUESITI A RISPOSTA APERTA

3 Come vengono gestiti gli *array* in Java?

4 Che caratteristiche deve avere il costruttore di copia di un *array* di oggetti?

5 Che cos'è il meccanismo *try/catch*?

6 Che cosa sono le eccezioni?

7 Che cos'è il meccanismo *handle or declare*?

8 Qual è la differenza tra le parole chiave *throws* e *throw*?

TEST A RISPOSTA MULTIPLA

9 In Java gli *array* sono oggetti?

A No, mai.
B Sì, in ogni caso.
C Sì, ma solo se non contengono tipi di dato primitivi.
D Sì, ma solo se contengono tipi di dato primitivi.

10 Un elemento di un *array* di oggetti è vuoto se:

A non contiene nulla.
B contiene il valore 0.
C contiene il valore predefinito `null`.
D contiene il riferimento a un oggetto creato con il costruttore di default.

11 Con l'istruzione `int[] unVettore;`:

A si istanzia un vettore di valori numerici interi.
B si dichiara un riferimento a un vettore di valori numerici interi.
C si istanzia un vettore di oggetti di tipo `int`.
D si dichiara un vettore di riferimenti a oggetti di tipo `int`.

12 Ponendo uguale a `null` un elemento di un *array* di riferimenti a oggetti:

A l'oggetto a cui faceva riferimento viene distrutto.
B l'oggetto a cui faceva riferimento viene distrutto solo se non è riferito da altre variabili.
C continua a esistere in ogni caso.
D per l'oggetto a cui faceva riferimento viene istanziata una variabile autonoma che lo riferisce.

13 L'attibuto *length* di un *array* contiene il numero di:

A elementi dell'*array*.
B elementi non vuoti dell'*array*.
C elementi vuoti dell'*array*.
D byte occupato dall'*array* nell'area di memoria *heap*.

14 Le eccezioni sono:

A eventi anomali rilevati dal compilatore in fase di generazione del *byte-code*.
B eventi anomali rilevati dall'ambiente di esecuzione a *run-time*.
C avvisi normalmente generati dall'ambiente di esecuzione.
D eventi anomali che vengono segnalati a *run-time* ma che comunque non pregiudicano l'esecuzione di un programma.

15 Le gestione delle eccezioni avviene in Java tramite il costrutto:

A `if/else`.
B `try/catch`.
C `switch/case`.
D `do/while`.

16 Indicare quali delle seguenti affermazioni sono vere rispetto alla generazione delle eccezioni.

A Se un metodo genera un'eccezione deve anche gestirla.
B Un metodo che genera un'eccezione deve contenere la clausola `throws` nella sua firma.
C Un'eccezione può essere generata da un metodo utilizzando l'istruzione `throw`.
D Un'eccezione può essere generata da un metodo utilizzando l'istruzione `try`.

17 **Indicare quali delle seguenti affermazioni sono false rispetto al sollevamento delle eccezioni.**

A L'istruzione `throw` sostituisce il costrutto `try/catch`.

B Nel codice di un metodo che genera eccezioni non è necessaria l'istruzione `return`.

C Per generare specifici tipi di eccezioni definite dall'utente è necessario definire una classe *ad hoc*.

D Un metodo non può generare più di un tipo di eccezioni.

18 **Indicare quali delle seguenti affermazioni circa le eccezioni sono vere.**

A Le eccezioni non necessariamente sono oggetti.

B Tutte le classi definite per rappresentare le eccezioni derivano gerarchicamente dalla classe Java *Trowable*.

C Tutte le classi definite per rappresentare le eccezioni derivano gerarchicamente dalla classe Java *Exception*.

D Tutte le classi definite per rappresentare le eccezioni derivano gerarchicamente dalla classe Java *RuntimeException*.

19 **La regola «*handle or declare*» si riferisce al fatto che:**

A tutti gli oggetti per essere gestiti (*handle*) devono essere preventivamente dichiarati (*declare*).

B a fronte di una possibile eccezione un metodo deve gestirla (*handle*), oppure dichiarare a sua volta di generarla (*declare*).

C a fronte di una possibile eccezione un metodo deve gestirla (*handle*) e dichiarare di generarla (*declare*).

D esiste un meccanismo omonimo per la gestione automatica delle eccezioni.

20 **Dati i metodi *insert1* e *insert2* aventi come scopo l'inserimento di un valore numerico intero in una specifica posizione del vettore *vector*, descritto dal frammento di codice riportato sotto, indicare quali delle seguenti affermazioni sono vere.**

A Il primo metodo genera un'eccezione, mentre il secondo no.

B Il primo metodo non genera un'eccezione, mentre il secondo sì.

C Il primo metodo intercetta un'eccezione, mentre il secondo no.

D Il primo metodo non intercetta un'eccezione, mentre il secondo sì.

```
...
private int[] vector;
...
public void insert1(int value, int position) throws ArrayIndexOutOfBoundsException {
    vector[position] = value;
    return;
}
...
public boolean insert2(int value, int position) {

      try {
            vector[position] = value;
            return true;
      }
      catch (ArrayIndexOutOfBoundsException exception) {
            return false;
      }
}
...
```

21 Serializzare un oggetto significa:

A memorizzarlo di seguito ad altri oggetti dello stesso tipo.

B memorizzarlo di seguito ad altri oggetti non necessariamente dello stesso tipo.

C salvare un oggetto su un supporto di memorizzazione sequenziale per renderlo persistente, o trasmetterlo su un canale di comunicazione sequenziale per renderlo trasferibile.

D memorizzarlo in parallelo ad altri oggetti dello stesso tipo.

22 Quali delle seguenti affermazioni sono vere rispetto al processo di serializzazione/deserializzazione di un oggetto?

A Permette di rendere un oggetto persistente.

B La deserializzazione permette di ripristinare un oggetto precedentemente serializzato.

C Una classe i cui oggetti devono essere serializzati deve implementare l'interfaccia Java *Serializable*.

D Il formato della serializzazione può essere sia binario sia testuale.

PROBLEMI

23 Facendo riferimento alla classe *Programma* progettata e implementata nell'esercizio 35 del capitolo precedente, progettare mediante un diagramma UML e implementare in linguaggio Java una classe i cui oggetti rappresentano dei contenitori, ciascuno dei quali può contenere fino a un massimo di *n* oggetti di tipo *Programma*. Dopo aver stabilito le strutture dati necessarie, si definiscano i seguenti metodi:

- costruttore avente come parametro il numero *n*;
- *getProgramma*: ha come parametro la posizione del programma nel contenitore e restituisce un oggetto *Programma* corrispondente a quello contenuto nella posizione specificata;
- *setProgramma*: inserisce un programma in una specifica posizione del contenitore e ha come parametro un oggetto *Programma*;
- *killProgramma*: elimina il programma presente nella posizione specificata come parametro;
- *getN*: restituisce il numero di programmi presenti nel contenitore;
- *cercaProgrammaPerDenominazione*: restituisce la posizione nel contenitore del programma con la denominazione corrispondente se presente, −1 altrimenti;
- *toString*: restituisce una stringa contenente l'elenco delle denominazioni di tutti i programmi contenuti nel contenitore;
- *confrontaContenitore*: consente di confrontare i programmi di un contenitore con quelli di un diverso contenitore restituendo il numero di programmi in comune.

24 Facendo riferimento alla classe *CD* progettata e implementata nell'esercizio 36 del capitolo precedente, progettare mediante un diagramma UML e implementare in linguaggio Java una classe *PortaCD* i cui oggetti rappresentano dei contenitori, ciascuno dei quali può contenere fino a un massimo di *n* oggetti di tipo *CD*. Dopo aver stabilito le strutture dati necessarie, si definiscano i seguenti metodi:

- costruttore avente come parametro il numero *n*;
- *getCD*: ha come parametro la posizione del CD nel portaCD e restituisce un oggetto *CD* corrispondente a quello contenuto nella posizione specificata;
- *setCD*: inserisce un CD in una specifica posizione del portaCD e ha come parametro un oggetto *CD*;
- *killCD*: elimina il CD presente nella posizione specificata del portaCD come parametro;
- *getN*: restituisce il numero di CD presenti nel portaCD;
- *cercaCDperTitolo*: restituisce la posizione nel portaCD del CD con il titolo corrispondente se presente, −1 altrimenti;
- *toString*: restituisce una stringa contenente l'elenco dei titoli di tutti i CD contenuti nel portaCD;
- *confrontaCollezione*: consente di confrontare i CD di un portaCD con quelli di una diversa collezione restituendo il numero di CD in comune.

25 Modificare la classe *Mensola* presentata nel corso del capitolo per fare in modo che, ferma restando la capienza massima, i libri siano sempre disposti in modo compatto dalla parte sinistra della mensola senza spazi vuoti tra un libro e l'altro:

A. inserire un libro in una determinata posizione comporta spostare tutti i libri a partire da tale posizione di un posto a destra;

B. inserire un libro in una posizione a destra dell'ultimo libro presente comporta inserire comunque il nuovo libro nella posizione immediatamente a destra dell'ultimo libro;

C. eliminare un libro in una determinata posizione significa spostare di una posizione verso sinistra tutti i libri alla sua destra.

26 Un albergo ha un sistema di gestione delle chiavi delle camere automatizzato: i clienti prendono e restituiscono le chiavi da un sistema portachiavi, senza l'intervento del portiere. Ogni chiave è identificata dal numero della camera (non necessariamente progressivo) ed è associata al nominativo del cliente a cui è stata assegnata la camera. Il sistema portachiavi può contenere le chiavi di tutte le camere dell'albergo e ogni posizione può essere occupata con una qualsiasi chiave. Il cliente che esce dall'albergo lascia la propria chiave nella prima posizione libera, il cliente che entra nell'albergo richiede la chiave fornendo il numero della camera o il proprio nominativo. Si richiede di rappresentare la soluzione mediante un diagramma UML delle classi e di implementarlo in linguaggio Java prevedendo le necessarie eccezioni.

27 Scrivere un metodo statico che – a partire dai valori dei coefficienti *a*, *b* e *c* di un'equazione di secondo grado – calcoli la prima soluzione generando una specifica eccezione nel caso di equazione impossibile.

28 La seguente classe Java implementa un semplice *stack* di interi:

```
public class Stack {
  private int[] mem;
  private int p,n;

  public Stack(int N) {
    mem = new int[N];
    p = 0;
    n = 0;
  }

  public void push(int x) {
    mem[p] = x;
    p++;
    n++;
  }

  public int pop() {
    int x;

    x = mem[p-1];
    p--;
    n--;
    return x;
  }

  public int size() {
    return n;
  }
}
```

Riscrivere i metodi *pop* e *push* della classe *Stack* in modo che sollevino specifiche eccezioni rispettivamente per la situazione di *stack* vuoto (*empty*) e pieno (*full*).

29 A SCUOLA DI LAVORO Una catena di autonoleggio deve gestire con un sistema informatico i propri veicoli; per ogni veicolo devono essere memorizzate le seguenti informazioni: codice, targa, marca e modello, numero di posti (si veda in proposito l'esercizio 37 del capitolo precedente). Si intende progettare una possibile soluzione per la gestione informatica di quasi 1000 veicoli avente le seguenti funzionalità:

- aggiunta di un nuovo veicolo (il codice deve essere un numero sequenziale incrementato automaticamente ogni volta che si aggiunge un veicolo);
- eliminazione di un veicolo dati il codice o la targa;
- ricerca delle informazioni di un veicolo dati il codice o la targa;
- ricerca di tutti i veicoli aventi un dato numero di posti;
- salvataggio e ripristino su/da file dell'intero insieme di veicoli;
- effettuare l'inventario di quante macchine per ogni marca dispone l'autonoleggio, nella forma marca, numero veicoli.

A. Definire mediante un diagramma UML le classi che consentono di rappresentare adeguatamente la soluzione del problema.

B. Implementare in linguaggio Java la classi progettate prevedendo e sollevando specifiche eccezioni.

C. Scrivere un metodo *main* che consenta la

gestione (aggiunta, eliminazione, ricerca per targa o per codice, elenco dato il numero di posti) dell'intero insieme di veicoli, visualizzando messaggi di errore in caso di sollevamento di eccezioni.

30 A SCUOLA DI LAVORO Un porto turistico affitta i propri posti-barca (circa un centinaio) alle imbarcazioni che ne fanno richiesta. Per legge è tenuto a registrare per ogni barca ospitata le seguenti informazioni: nome, nazionalità, lunghezza, stazza, tipologia (vela o motore); ma non vi è obbligo di mantenere le informazioni relative alle imbarcazioni dopo che hanno lasciato il porto. I posti-barca sono numerati: i posti da 1 a 20 non possono ospitare barche più lunghe di 10 m e le barche a vela devono essere piazzate in via prioritaria nei posti successivi al 50. Il costo dell'affitto per le barche a vela è di 10 € per metro di lunghezza al giorno, mentre per le barche a motore è di 20 € per tonnellata di stazza al giorno. È richiesta la progettazione di una possibile soluzione per la gestione informatica dei posti-barca che implementi le seguenti funzionalità:

- assegnazione di un posto a una barca in arrivo;
- liberazione di un posto occupato con calcolo dell'importo dell'affitto (in input viene fornito il numero dei giorni di sosta);
- ricerca delle informazioni relative alla barca che occupa un dato posto;
- salvataggio su file dello stato del porto in un certo istante in modo da renderlo persistente;
- produrre una struttura dati (*array*) dei nomi delle barche di una certa nazionalità specificata dall'utente.

A. Definire mediante un diagramma UML le classi che consentono di rappresentare adeguatamente la soluzione del problema.

B. Implementare in linguaggio Java le classi progettate prevedendo e sollevando specifiche eccezioni.

C. Dotare la classe principale di un metodo *main* che permetta all'utente di esercitare le funzionalità elencate visualizzando messaggi di errore in caso di sollevamento di eccezioni.

31 A SCUOLA DI LAVORO Una biblioteca scolastica deve gestire mediante un'applicazione software un elenco di circa 1000 libri: per ogni libro è necessario memorizzare l'autore, il titolo, l'anno di pubblicazione e l'editore. L'applicazione deve consentire le seguenti operazioni:

- aggiunta di un nuovo libro alla biblioteca;
- ricerca di un libro a partire dal titolo;
- ricerca di tutti i libri di uno specifico autore;
- determinazione del numero di libri presenti.

Si richiede di rappresentare la soluzione mediante un diagramma UML delle classi e di implementarla in linguaggio Java prevedendo la generazione delle opportune eccezioni.

32 A SCUOLA DI LAVORO Una grande organizzazione deve catalogare i computer utilizzati dai propri dipendenti; per ogni computer devono essere memorizzate le seguenti informazioni: codice, marca, modello, velocità del processore, dimensioni della memoria RAM, dimensioni del disco, dimensioni del monitor e anno di acquisto. Il codice di ogni computer è un numero progressivo generato automaticamente e non ulteriormente modificabile. Dopo aver definito mediante un diagramma UML e implementato in linguaggio Java una classe per rappresentare gli oggetti di tipo *Computer* (si veda in proposito l'esercizio 38 del capitolo precedente), progettare in linguaggio UML e implementare in linguaggio Java una classe che consenta la gestione (aggiunta, eliminazione, ricerca per codice, ricerca di tutti i computer con caratteristiche migliori di valori di velocità e dimensione di memoria/disco specificate, salvataggio e ripristino su/da file dell'intero catalogo), prevedendo e generando specifiche eccezioni.

CASI PRATICI

33 Si vuole simulare un cronometro che effettui la misura del tempo in secondi. Realizzare un diagramma UML e la relativa implementazione in linguaggio Java di una classe *Cronometro* assumendo che le operazioni possibili siano le seguenti:

`init()` inizializza un oggetto cronometro azzerando il suo accumulatore di secondi

`start()` avvia il cronometro che inizia a contare i secondi

`stop()` ferma il cronometro interrompendo il conteggio dei secondi

`show()` prevede un parametro che può assumere i valori «s», «m» o «h» in funzione del quale visualizza il tempo conteggiato in secondi dal cronometro rispettivamente in: secondi, minuti:secondi, ore:minuti:secondi

Per realizzare la classe *Cronometro* si utilizzi il metodo statico *System.currentTimeMillis*, che restituisce un valore di tipo `long` che rappresenta il numero di millisecondi trascorsi dalle ore 00:00 del 1/1/1970.
Inoltre, dotare la classe di un costruttore che azzeri il cronometro e di un metodo *main* che istanzi due oggetti di tipo *Cronometro c1* e *c2* e che visualizzi i risultati forniti dalla seguente sequenza di operazioni:

- *start* di *c1*
- *attesa 5 secondi*
- *show* in secondi del valore di *c1*
- *start* di *c2*
- *attesa 61 secondi*
- *stop* di *c1*
- *show* in minuti:secondi di *c1*
- *attesa 4 secondi*
- *start* di *c1*
- *attesa 2 secondi*
- *stop* di *c2*
- *show* di *c2* in secondi
- *stop* di *c1*
- *show* di *c1* in ore:minuti:secondi

Per realizzare il metodo *main* si implementi il seguente metodo statico avente lo scopo di sospendere l'esecuzione del codice di *n* secondi:

```
public static void attendiNsecondi
   (long n) {
   try {
         Thread.currentThread().sleep
         (n * 1000);
   }
   catch (InterruptedException exception) {
   }
}
```

34 Facendo riferimento alla classe *Vettore* implementata nell'esercizio 42 del capitolo precedente, aggiungere un metodo booleano *interseca* avente come parametro un altro oggetto *Vettore* e che restituisca `true` se i due vettori si intersecano, `false` altrimenti. Per implementare il metodo *interseca* si prenda in considerazione il codice C/C++ riportato nella pagina seguente.

Nei sistemi CAD (*Computer Aided Design*) e GIS (*Geographic Information System*) una *polyline* è una sequenza ordinata di *n* vettori *v1, v2, v3, ..., vn* nel piano cartesiano concatenati in modo che l'origine di ciascuno (escluso il primo) coincida con il vertice del precedente, come illustrato negli esempi seguenti che riproducono rispettivamente una *polyline* intrecciata e una semplice:

Progettare – mediante un diagramma di classe UML – e implementare in linguaggio Java una classe *Polyline* che consenta di definire in sequenza tutti i vettori che costituiscono la *polyline* stessa e inoltre di eseguire le seguenti operazioni:

- calcolare la lunghezza totale della *polyline*;
- determinare se la *polyline* è semplice o intrecciata;
- salvare e ripristinare tutti i vettori che costituiscono la *polyline* in/da un file di tipo testuale.

35 Si intende realizzare un programma Java che consenta di gestire la navigazione in barca. Il programma deve permettere all'operatore di inserire manualmente le successive posizioni rilevate mediante un dispositivo GPS, che restituisce latitudine e longitudine in gradi, primi e secondi, e di calcolare in ogni momento la distanza percorsa dall'ultimo punto rilevato e dal punto di partenza. Il programma deve permettere inoltre di inserire una destinazione espressa mediante i valori della latitudine e della longitudine e di calcolare la rotta (cioè l'angolo rispetto al Nord) e la lunghezza del percorso rettilineo che congiunge l'ultimo punto rilevato con il punto di destinazione. Sono richiesti:

- un diagramma UML delle classi;
- un programma in linguaggio Java con interfaccia utente testuale.

Per adattare le coordinate geografiche (latitudine/longitudine) in formato decimale in coordinate cartesiane Nord/Est espresse in metri, modificare il codice C/C++ riportato a pagina A140.

```
struct POINT
   {
     float x;
     float y;
   };

struct SEGMENT
   {
     POINT p1;
     POINT p2;
   };

int ccw (POINT p0, POINT p1, POINT p2)
   {
     int dx1, dx2, dy1, dy2;

     dx1 = p1.x - p0.x;
     dy1 = p1.y - p0.y;
     dx2 = p2.x - p0.x;
     dy2 = p2.y - p0.y;

     if (dx1*dy2 > dy1*dx2)
       return +1;
     if (dx1*dy2 < dy1*dx2)
       return -1;
     if ((dx1*dx2 < 0) || (dy1*dy2 < 0))
       return -1;
     if ((dx1*dx1+dy1*dy1) < (dx2*dx2+dy2*dy2))
       return +1;
     return 0;
   }

bool intersect(SEGMENT s1, SEGMENT s2)
   {
     return ((ccw(s1.p1,s1.p2,s2.p1)*ccw(s1.p1,s1.p2,s2.p2)) <=0) &&
       ((ccw(s2.p1,s2.p2,s1.p1)*ccw(s2.p1,s2.p2,s1.p2)) <=0);
   }
```

[R. Sedgewick, *Algoritmi in C++*, Addison-Wesley, 1993]

```
#define PI                              3.141592653589793

#define WGS84_E2                        0.006694379990197
#define WGS84_E4                        WGS84_E2*WGS84_E2
#define WGS84_E6                        WGS84_E4*WGS84_E2
#define WGS84_SEMI_MAJOR_AXIS           6378137.0
#define WGS84_SEMI_MINOR_AXIS           6356752.314245
#define UTM_LONGITUDE_OF_ORIGIN         3.0/180.0*PI
#define UTM_LATITUDE_OF_ORIGIN          0.0
#define UTM_FALSE_EASTING               500000.0
#define UTM_FALSE_NORTHING_N            0.0
#define UTM_FALSE_NORTHING_S            10000000.0
#define UTM_SCALE_FACTOR                0.9996

double m_calc(double latitude)
{
  return (1.0 - WGS84_E2/4.0 -
          3.0*WGS84_E4/64.0 -
          5.0*WGS84_E6/256.0) * latitude -
          (3.0*WGS84_E2/8.0 +
           3.0*WGS84_E4/32.0 +
           45.0*WGS84_E6/1024.0) * sin(2.0*latitude) +
          (15.0*WGS84_E4/256.0 +
           45.0*WGS84_E6/1024.0) * sin(4.0*latitude) -
          (35.0*WGS84_E6/3072.0) * sin(6.0*latitude);
}

// INPUT: position in latitude/longitude (WGS84)
// OUTPUT: position in UTM easting/northing (meters)
void GPS2UTM(double latitude, double longitude, double* easting, double* northing)
{
  int int_zone;
  double M, M_origin, A, A2, e2_prim, C, T, v;

  int_zone = (int)(longitude/6.0);
  if (longitude < 0)
    int_zone--;
  longitude -= (double)(int_zone)*6.0;
  longitude *= PI/180.0;
  latitude *= PI/180.0;
  M = WGS84_SEMI_MAJOR_AXIS*m_calc(latitude);
  M_origin = WGS84_SEMI_MAJOR_AXIS * m_calc(UTM_LATITUDE_OF_ORIGIN);
  A = (longitude - UTM_LONGITUDE_OF_ORIGIN) * cos(latitude);
  A2 = A*A;
  e2_prim = WGS84_E2/(1.0 - WGS84_E2);
  C = e2_prim*pow(cos(latitude),2.0);
  T = tan(latitude);
  T *= T;
  v = WGS84_SEMI_MAJOR_AXIS /
      sqrt(1.0 - WGS84_E2 * pow(sin(latitude),2.0));
  *northing = UTM_SCALE_FACTOR*(M - M_origin + v*tan(latitude)*
              (A2/2.0 + (5.0 - T + 9.0*C + 4.0*C*C)*
              A2*A2/24.0 + (61.0 - 58.0*T + T*T + 600.0*C -
              330.0*e2_prim)*A2*A2*A2/720.0));
  if (latitude < 0)
    *northing += UTM_FALSE_NORTHING_S;
  *easting = UTM_FALSE_EASTING + UTM_SCALE_FACTOR*v *
             (A + (1.0 - T + C)*A2*A/6.0 +
               (5.0 - 18.0*T + T*T + 72.0*C - 58.0*e2_prim)*
             A2*A2*A/120.0);
  return;
}
```

A4 Strutture dati

Si deve realizzare un programma per la gestione dell'elenco degli invitati a una festa dove il numero di questi è variabile: a seconda delle necessità dovremo aggiungere nuove persone a cui inizialmente non avevamo pensato, oppure eliminarne altre che hanno declinato l'invito. In casi di questo tipo l'utilizzo di un *array* non offre la necessaria flessibilità: infatti fissare a priori un determinato numero di elementi può comportare un eccesso di elementi inutilizzati, oppure trovarsi senza posizioni disponibili se il numero degli elementi cresce oltre il previsto.

È necessario realizzare una struttura dati di dimensione variabile in cui sia possibile inserire o eliminare elementi in una posizione qualsiasi. Schematicamente una struttura dati di questo tipo può essere rappresentata come in FIGURA 1, dove:

- *head* («testa») è il riferimento d'ingresso alla struttura;
- ogni elemento, denominato «nodo», della struttura è un oggetto ripartito logicamente in due componenti: una informativa (*info*), che in questo caso contiene il riferimento ai dati di un *Invitato* (*Invitato 1*, *Invitato 2*, ... *Invitato n*), e una che realizza il collegamento all'*Invitato* successivo (*link*);
- l'ultimo nodo ha la componente *link* con valore `null`;
- per scorrere la lista è necessario seguire la catena dei riferimenti (l'accesso è strettamente sequenziale);
- è possibile inserire o eliminare nodi alla lista inserendoli/eliminandoli in una posizione qualsiasi (all'inizio, al termine, in una posizione intermedia) modificando in modo opportuno le componenti *link* dei nodi.

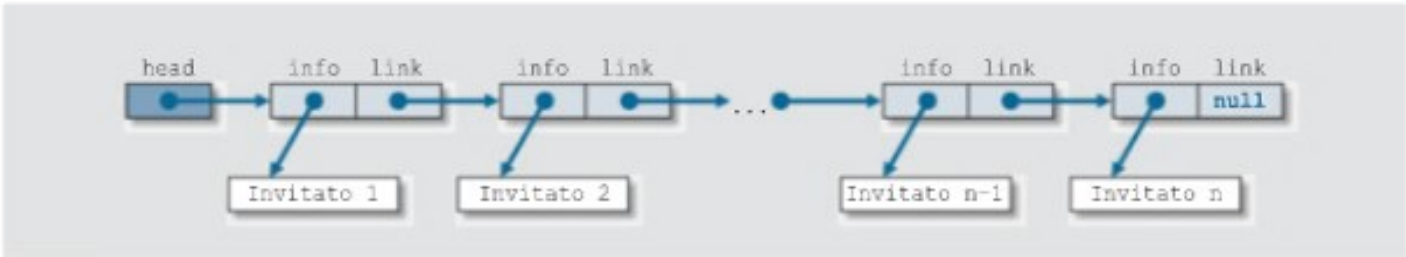


FIGURA 1

Per inserire, per esempio, un nuovo invitato come secondo elemento della lista (magari per collocarlo vicino a un altro invitato che conosce) si opera come segue (FIGURA 2):

1. Si crea un nuovo nodo la cui componente *info* riferisce il *Nuovo Invitato*.
2. Si imposta la componente *link* del nuovo nodo in modo che riferisca al nodo relativo a *Invitato 2*.
3. Si imposta la componente *link* del nodo relativo a *Invitato 1* in modo che riferisca al nodo relativo a *Nuovo Invitato*, eliminando di fatto il preesistente collegamento tra i nodi relativi a *Invitato 1* e *Invitato 2*.

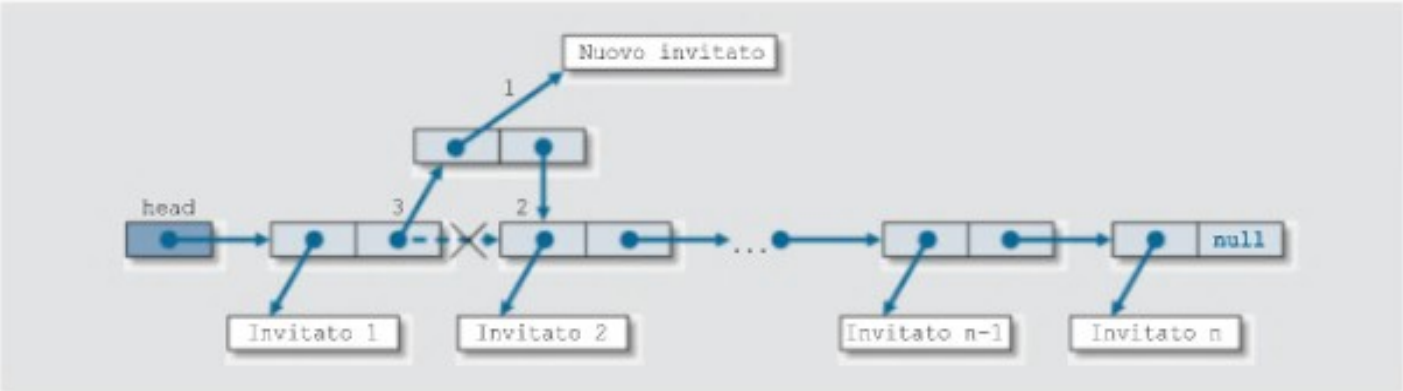


FIGURA 2

Strutture dati come quella appena descritta prendono il nome di **liste**.

Il diagramma delle classi UML della lista degli invitati potrebbe essere quello mostrato in FIGURA 3.

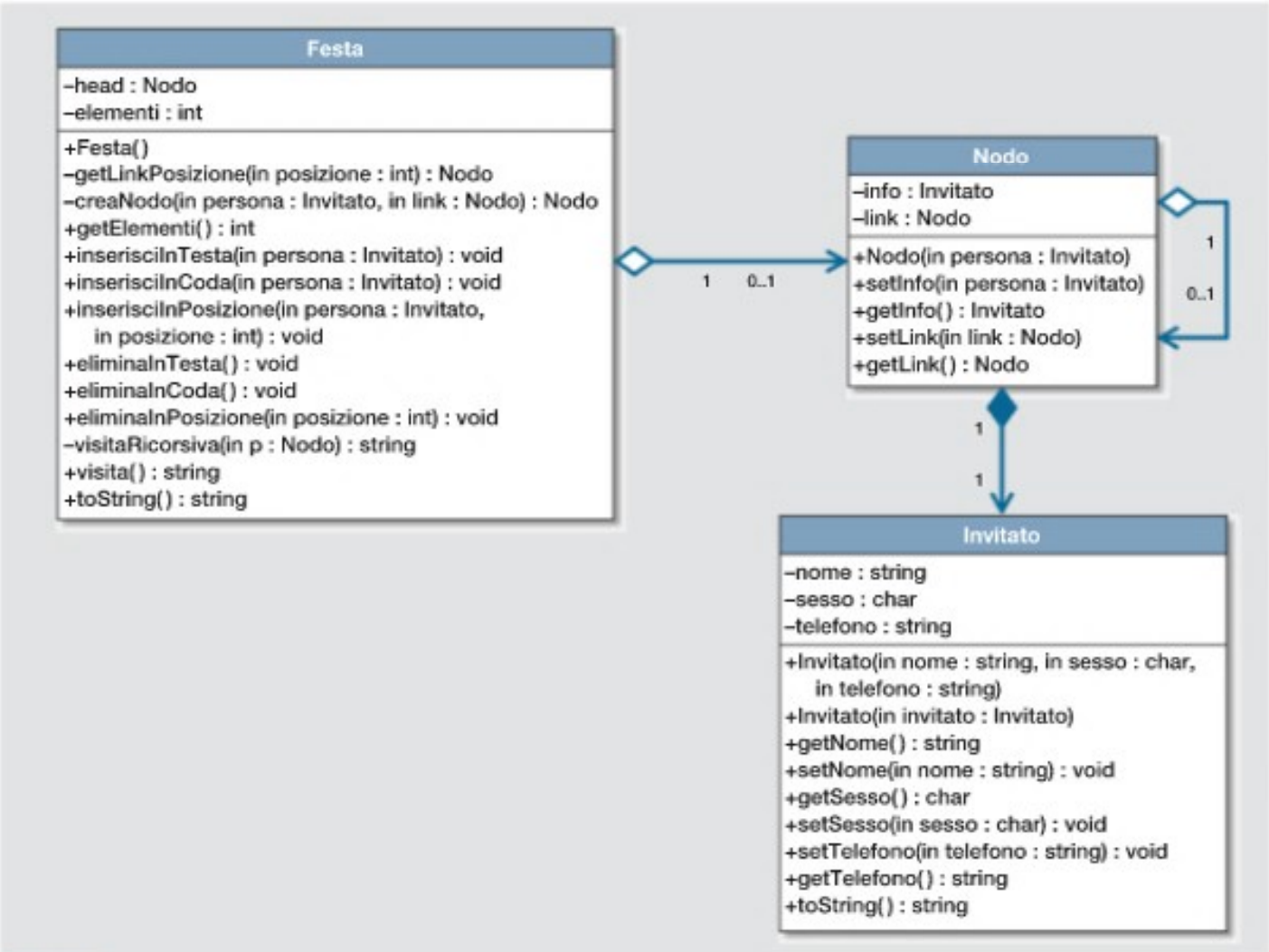


FIGURA 3

OSSERVAZIONE Nel diagramma UML precedente è evidente che ogni oggetto di classe *Nodo* ha:

- un attributo *info* (la componente informativa) che riferisce un oggetto di classe *Invitato*;
- un attributo *link* di classe *Nodo* che realizza il collegamento con un altro oggetto dello stesso tipo allo scopo di costituire l'intera sequenza.

Gli oggetti di classe *Festa* – implementata mediante una lista semplice – hanno come attributi:

- il riferimento al nodo iniziale *head* della lista;
- il numero di nodi che in un determinato istante sono concatenati nella lista.

1 Implementazione di una lista in linguaggio Java

L'*array* è il tipo di dato strutturato che tutti i linguaggi di programmazione rendono normalmente disponibile: le informazioni che esso contiene assumono la forma di un elenco di dati singolarmente accedibili in base a un indice che ne rappresenta la posizione all'interno dell'*array* stesso. Esso è semplice da essere usato, ma presenta una scarsa flessibilità:

- è in genere caratterizzato da un numero fisso di elementi che, una volta definito, non può essere modificato;
- nel caso di elenchi ordinati, come per esempio nomi da mantenere in ordine alfabetico, l'inserimento di un nuovo elemento comporta lo spostamento di una posizione di tutti i nomi che lo seguono; un'operazione analoga è richiesta per l'eliminazione di un elemento.

Inserimento ed eliminazione di un elemento da un elenco possono essere realizzati in modo più efficiente utilizzando una struttura in cui i dati siano memorizzati in elementi distinti a cui sia possibile applicare un ordinamento logico arbitrario diverso dalla loro posizione fisica: la **lista**.

Graficamente una lista viene rappresentata nel modo visto nel paragrafo precedente: ogni elemento è collegato al successivo mediante un riferimento – schematizzato mediante una freccia – e l'accesso all'insieme degli elementi è dato dal riferimento iniziale denominato ***head***. È sulla base della catena dei riferimenti che si ha la visione di contiguità logica degli elementi dell'insieme, i quali possono occupare fisicamente posizioni di memoria qualsiasi senza avere alcun ordinamento precostituito.

Nella TABELLA 1 viene riassunto il confronto tra *array* e lista semplice.

TABELLA 1

	Vantaggi	Svantaggi
Array	• Semplicità di gestione. • Accesso diretto agli elementi.	• Gestione statica del numero degli elementi. • Difficoltà delle operazioni di inserimento e cancellazione di elementi con criteri di ordinamento.
Lista	• Gestione dinamica del numero degli elementi. • Facilità nelle operazioni di inserimento e cancellazione di elementi anche in presenza di criteri di ordinamento.	• Maggiore complessità di gestione rispetto all'*array*. • Struttura ad accesso strettamente sequenziale.

In generale una lista viene implementata nel linguaggio di programmazione Java definendo una classe *Nodo* (*Tipo* rappresenta in questo caso la classe della componente informativa):

```
public class Nodo {
  private Tipo info;
  private Nodo link;

  public Nodo (Tipo oggetto) {
    info = new Tipo(oggetto);
    link = null;
  }

  public void setInfo(Tipo oggetto) {
    info = new Tipo(oggetto);
  }

  public Tipo getInfo() {
    return new Tipo(info);
  }

  public void setLink(Nodo link) {
    this.link = link;
  }

  public Nodo getLink() {
    return link;
  }
}
```

La classe *Lista* vera e propria avrà la seguente struttura di base:

```
public class Lista {
  private Nodo head;      // riferimento al nodo iniziale
                          // della lista
  private int elementi;   // numero di elementi (nodi) della
                          // lista

  public Lista() {
    head = null;
```

```
    elementi = 0;
  }

  ...
  ...
  ...
}
```

1.1 Operazioni su una lista

Le operazioni di *visita*, *inserimento* ed *eliminazione* degli elementi di una lista sono effettuate lavorando sui riferimenti che collegano tra di loro gli elementi.

Per semplicità, nel presentare le operazioni sulle liste, si farà riferimento solo alla componente che attiene alla catena dei nodi, che costituisce l'essenza della lista, non esplicitando la componente relativa agli oggetti di qualunque tipo che costituiscono l'informazione associata ai nodi.

Visita degli elementi di una lista

L'algoritmo che permette di scorrere tutti gli elementi di una lista può essere implementato con il metodo

```
public void visitaLista() {
  Nodo p = head;

  while(p != null) {
    esamina(p.getInfo());
    p = p.getLink();
  }
}
```

dove:

- la prima istruzione assegna alla variabile *p* utilizzata per scorrere gli elementi della lista il valore del riferimento iniziale della lista (*head*):

- il ciclo procede fino a quando *p* assume un valore diverso da `null` (lista non ancora terminata);
- *esamina(p.getInfo())* rappresenta l'invocazione di un generico metodo da applicare all'informazione dell'elemento riferito da *p* (per esempio la visualizzazione del suo contenuto);
- l'istruzione *p = p.getLink();* serve a far avanzare il riferimento *p* al nodo successivo della lista:

Quando *p* arriva a riferire l'ultimo elemento della lista

l'esecuzione di questa istruzione fa sì che *p* assuma il valore dell'attributo *link* dell'ultimo nodo, ovvero `null`, e il ciclo termina.

> L'attributo *head* deve essere gestito in maniera accorta: perdendo il riferimento al primo nodo, non vi è più modo di accedere a tutti i nodi della lista. Allo stesso modo, perdendo il valore *link* di uno qualsiasi dei nodi della lista, si rendono inaccessibili tutti i nodi della lista successivi.

La funzione di visita può essere implementata anche in modo ricorsivo:

```
private void visitaRicorsivaLista(Nodo p) {

  if (p == null)
    return;
  esamina(p.getInfo());
  visitaRicorsivaLista(p.getLink());
}
```

L'invocazione di questo metodo privato può essere effettuata da un metodo pubblico come il seguente:

```
public void visitaLista() {
  visitaRicorsivaLista(head);
}
```

Inserimento di un nuovo elemento

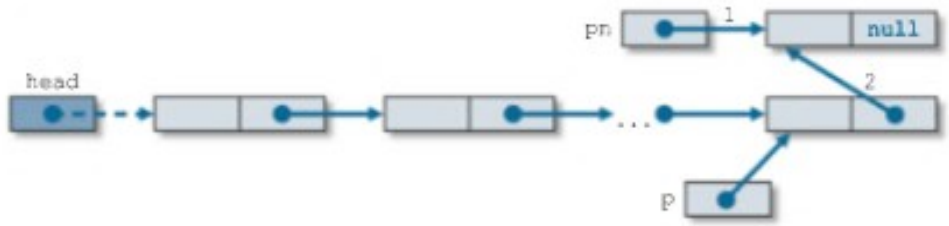
L'inserimento di un elemento richiede procedimenti diversi in funzione del punto di inserimento: in testa, in coda, o in una posizione intermedia.

Nel caso di **inserimento in testa** di un nuovo nodo l'algoritmo è piuttosto semplice:

```
public void inserisciInTesta() {
  Nodo pn = new Nodo();  // creazione nuovo nodo riferito da pn

  pn.setLink(head);      // la componente link di pn riferisce
                         // head
  head = pn;             // pn diviene il nuovo primo nodo
  elementi++;            // aggiornamento del numero di nodi
}
```

Per l'**inserimento in coda** è necessario, dopo aver creato il nuovo elemento, scorrere la lista posizionandosi sull'ultimo nodo mediante una variabile di riferimento e collegare a esso il nuovo nodo:

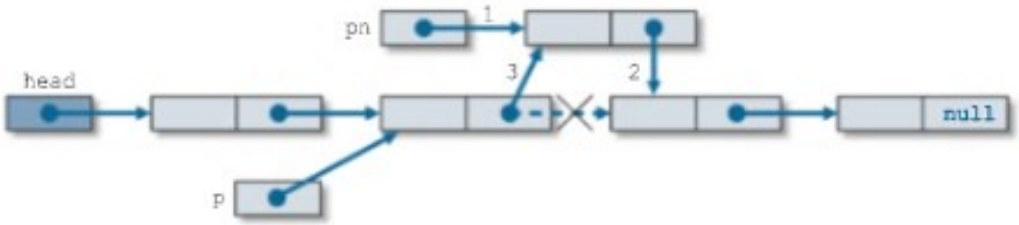
```
public void inserisciInCoda() {
  Nodo p = head; // riferimento per scorrere la lista
  Nodo pn = new Nodo(); // creazione nuovo nodo

  if (p==null) // lista vuota?
    inserisciInTesta();
  else {
    while (p.getLink()!= null) // scorrimento lista
      p = p.getLink();

    pn.setLink(null);  // il nuovo nodo è l'ultimo della lista
    p.setLink(pn);     // pn diviene il nuovo ultimo nodo
    elementi++;        // aggiornamento del numero di nodi
  }
}
```



Si noti che il test di fine ciclo è *p.getLink()!=* `null` e non *p!=* `null` perché in quest'ultimo caso il ciclo si arresterebbe con *p* uguale a `null` e pertanto *p* non riferirebbe più alcun nodo: saremmo già usciti dalla lista! Inoltre, se la lista è vuota (*head* è uguale a `null`), l'inserimento di un nodo in coda equivale all'inserimento di un nodo in testa perché deve essere aggiustato anche il riferimento iniziale alla lista *head*.

Per l'**inserimento in una posizione intermedia** è necessario creare il nuovo elemento, scorrere la lista fino al nodo precedente la posizione di inserimento tramite una variabile riferimento e aggiustare i vari collegamenti come mostrato di seguito:

```
public void inserisciInPosizione(int posizione) {
  Nodo p = head; // riferimento per scorrere la lista
  Nodo pn = new Nodo(); // creazione nuovo nodo
  int i = 1; // indice di posizione

  if ((posizione<=1) || (head==null))
    inserisciInTesta();
  else
    if (posizione>elementi)
      inserisciInCoda();
    else {
      while(i<posizione-1) // scorrimento lista fino alla
                           // posizione
        {p=p.getLink();i++;};
      pn.setLink(p.getLink()); // pn riferisce il nodo
                               // successivo
      p.setLink(pn);           // il nuovo nodo viene collegato
      elementi++;              // aggiornamento del numero di
                               // nodi
    }
}
```



Per scorrere la lista fino a individuare il punto di inserimento si è impiegato il ciclo

```
while(i<posizione-1) p=p.getLink();
```

che si interrompe con la variabile *p* che riferisce il nodo precedente il punto di inserimento (*posizione-1*). Nel caso in cui il criterio di inserimento del nodo non sia la posizione occupata, ma una condizione espressa sulla componente informativa (per esempio l'inserimento degli invitati nell'elenco in ordine alfabetico), è necessario utilizzare una tecnica basata su due riferimenti:

```
Nodo ps=head, pp=head;

while ((ps.getLink()!= null) && (...ps.getInfo()...) {
  pp = ps;
  ps = ps.getLink();
}
```

Il ciclo si interrompe se si raggiunge la fine della lista senza aver trovato l'elemento da esaminare (*ps.getLink()* restituisce valore `null`), oppure se l'elemento da esaminare è stato individuato (diviene falsa la condizione che coinvolge *ps.getInfo()*) .

Eliminazione di un elemento

Anche l'eliminazione di un elemento richiede procedimenti diversi in funzione del punto di eliminazione: in testa, in coda, in una posizione intermedia.

Nel caso dell'**eliminazione in testa**, l'operazione si riduce all'aggiornamento del riferimento iniziale e al decremento del numero degli elementi:

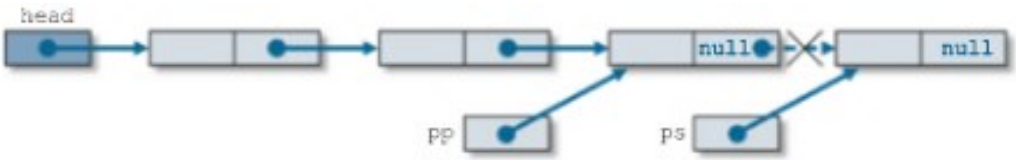
```
public void eliminaInTesta() {
  if (head==null) // lista vuota?
    return;
  head = head.getLink(); // head riferisce il secondo nodo
                         // della lista
  elementi--;            // aggiornamento del numero di nodi
}
```

Se la lista ha un solo elemento, *head* – che assume il valore della componente *link* del primo nodo che nel caso specifico ha valore `null` – diviene a sua volta `null`. È importante effettuare sempre un test per verificare se la lista è vuota (*head* uguale a `null`) prima di procedere con qual-

siasi operazione di eliminazione. Infatti, se la lista è vuota, l'istruzione *head = head.getLink()* genera un'eccezione perché la componente *link* non esiste e se non gestita causa l'interruzione anomala del programma. L'alternativa è tra gestire questo tipo di eccezioni o prevenirne la generazione verificando il valore del riferimento *head* per decidere le operazioni corrette da svolgere.

Per effettuare un'**eliminazione in coda** è necessario posizionarsi sul penultimo nodo mediante una variabile di riferimento e impostare al valore `null` la componente *link*:

```
public void eliminaInCoda() {
  Nodo ps=head, pp=head; // variabili per scorrere la lista

  if (pp==null) // lista vuota?
    return;
  while(ps.getLink() != null) { // scorrimento della lista
    pp = ps;
    ps = ps.getLink();
  }
  if (pp==head) // solo un nodo nella lista?
    eliminaInTesta();
  else {
    pp.setLink(null);  // il penultimo nodo diviene ultimo
  elementi--;          // aggiornamento del numero di nodi
  }
}
```



Per portare la variabile *pp* a riferire il penultimo nodo della lista è stata usata la seguente procedura dove, oltre a *pp*, è stata impiegata la variabile *ps*:

```
Nodo ps=head, pp=head;

while(ps.getLink()!= null) {
  pp = ps;
  ps = ps.getLink();
}
```

Nel ciclo *pp* riferisce sempre il nodo precedente rispetto a quello riferito da *ps*: quando *ps.getLink()*== `null` risulta vera, *ps* riferirà l'ultimo nodo e *pp* riferirà il penultimo nodo.
In alternativa si potrebbe utilizzare un metodo che, basandosi sul valore dell'attributo *elementi* (numero di nodi presenti nella lista) restituisca il riferimento di un nodo in posizione *n*-esima, oppure `null` se la posizione non è corretta:

Liste bidirezionali

Molte delle difficoltà degli algoritmi di gestione delle liste (in particolare relativamente alle operazioni di inserimento e di eliminazione) sono dovute alla necessità di scorrere la lista fino all'ultimo elemento e all'uso di un doppio riferimento per tracciare il nodo precedente in una scansione.
Facendo in modo che la lista abbia una doppia concatenazione in cui ogni nodo, oltre a collegare il nodo successivo, colleghi anche il nodo precedente e aggiungendo un attributo di riferimento per l'ultimo nodo della lista, è possibile ridefinire molti degli algoritmi illustrati in modo molto più semplice.

```
private Nodo getLinkPosizione(int n) {
  Nodo p = head;
  int i = 1; // variabile usata come indice di posizione

  if ((n>elementi) || (n<1)) // posizione corretta?
    return null;
  else
    // scorrimento lista fino alla posizione n-esima
    while ((p.getLink()!= null) && (i<n)) {
      p = p.getLink();
      i++;
    }
  return p;
}
```

In questo modo, dato che l'invocazione del metodo *getLinkPosizione(n)* restituisce il riferimento del nodo nella *n*-esima posizione, per ottenere il riferimento al penultimo nodo sarà sufficiente l'istruzione *getLinkPosizione(elementi-1)*.

Per effettuare un'**eliminazione in posizione intermedia** è necessario, utilizzando una delle tecniche descritte in riferimento all'eliminazione in coda, posizionarsi sul nodo precedente alla posizione di eliminazione e aggiustare i vari collegamenti come mostrato di seguito:

```
public void eliminaInPosizione(int n) {
  Nodo pp, ps;

  if (n<=1)
    eliminaInTesta();
  else
    if (n>=elementi)
      eliminaInCoda();
    else {
      pp = getLinkPosizione(n-1);
      if (pp==null) // riferimento valido?
        return;
      ps = pp.getLink();          // ps riferisce il nodo
                                  // successore di pp
      pp.setLink(ps.getLink());   // pp riferisce il nodo
                                  // successore di ps
      elementi--;                 // aggiornamento del numero
                                  // di nodi
    }
  }
```

OSSERVAZIONE Nel caso in cui il criterio di individuazione del nodo da eliminare non sia la posizione ma una condizione espressa sulla componente informativa (per esempio un invitato di cui è dato il nominativo), è necessario utilizzare la tecnica di scorrimento sincronizzato dei due riferimenti *pp* e *ps*:

```
Nodo ps=head, pp=head;
while ((ps.getLink()!= null) && (...ps.getInfo()...) {
   pp = ps;
   ps=ps.getLink();
}
```

Il ciclo si interrompe se si raggiunge la fine della lista senza aver trovato l'elemento da eliminare (*ps.getLink()* diviene `null`), oppure se l'elemento da eliminare viene trovato (diviene falsa la condizione relativa a *ps.getInfo()*).

ESEMPIO

Vediamo un'implementazione completa in linguaggio Java dell'esempio presentato in apertura del capitolo con i metodi della classe *Festa* – che implementa la lista degli invitati – che istanziano e generano specifiche eccezioni in caso di errore:

```
public class Invitato {
   private String nome;
   private char sesso;
   private String telefono;

   public Invitato(String nome, char sesso, String telefono) {
      this.nome = nome;
      this.sesso = sesso;
      this.telefono = telefono;
   }

   public Invitato (Invitato invitato) {
      this.nome = invitato.getNome();
      this.sesso = invitato.getSesso();
      this.telefono = invitato.getTelefono();
   }

   public void setNome(String nome) {
      this.nome=nome;
   }

   public String getNome() {
      return nome;
   }

   public void setSesso(char sesso) {
      this.sesso=sesso;
   }

   public char getSesso() {
      return sesso;
   }

   public void setTelefono(String telefono) {
      this.telefono=telefono;
   }
```

```
    public String getTelefono() {
      return telefono;
    }

    public String toString() {
      return nome + "," + sesso + "," + telefono;
    }
  }

  public class Nodo {
    private Invitato info;
    private Nodo link;

    public Nodo (Invitato persona) {
      info = new Invitato(persona);
      link = null;
    }

    public void setInfo(Invitato persona) {
      info = new Invitato(persona);
    }

    public Invitato getInfo() {
      return new Invitato(info);
    }

    public void setLink(Nodo link) {
      this.link = link;
    }

    public Nodo getLink() {
      return link;
    }
  }

  public class FestaException extends Exception {
    private String error = "";

    FestaException(String error) {
      this.error = error;
    }

    String getError() {
      return error;
    }
  }

  public class Festa {
    private Nodo head;
    private int elementi;

    public Festa() {
      head = null;
      elementi=0;
    }
```

```
private Nodo getLinkPosizione(int posizione) throws FestaException {
  int n = 1;
  Nodo p = head;

  if (head==null)
    throw new FestaException("Lista vuota");
  if ((posizione>elementi) || (posizione<1))
    throw new FestaException("Posizione errata");
  while ((p.getLink()!= null) && (n<posizione)) {
    p = p.getLink();
    n++;
  }
  return p;
}

private Nodo creaNodo(Invitato persona, Nodo link) {
  Nodo nuovoNodo = new Nodo(persona);
  nuovoNodo.setLink(link);
  return nuovoNodo;
}

public int getElementi() {
  return elementi;
}

public void inserisciInTesta(Invitato persona) {
  Nodo p = creaNodo(persona, head);
  head = p;
  elementi++;
  return;
}

public void inserisciInCoda(Invitato persona) {
  if (head==null)
    inserisciInTesta(persona);
  else {
    try {
      Nodo p = getLinkPosizione(elementi);
      p.setLink(creaNodo(persona, null));
      elementi++;
    }
    catch (FestaException exception)
    {
    }
  return;
  }
}
public void inserisciInPosizione(Invitato persona, int posizione)
            throws FestaException {
  if (posizione<=1)
    inserisciInTesta(persona);
  else {
    if (elementi<posizione)
      inserisciInCoda(persona);
    else {
```

```
        Nodo p = getLinkPosizione(posizione-1);
        p.setLink(creaNodo(persona, p.getLink()));
        elementi++;
      }
    }
    return;
  }

  public void eliminaInTesta() throws FestaException {
    if (head==null)
      throw new FestaException("Lista vuota");
    head=head.getLink();
    elementi--;
    return;
  }

  public void eliminaInCoda() throws FestaException {
    if (head==null)
      throw new FestaException("Lista vuota");
    Nodo p=getLinkPosizione(elementi-1);
    p.setLink(null);
    elementi--;
    return;
  }

  public void eliminaInPosizione(int posizione)throws FestaException {
    if (posizione==1)
      eliminaInTesta();
    else
      if (posizione==elementi)
        eliminaInCoda();
      else {
        Nodo ps = getLinkPosizione(posizione);
        Nodo pp = getLinkPosizione(posizione-1);
        pp.setLink(ps.getLink());
        elementi--;
      }
    return;
  }

  private String visita(Nodo p) {
    if (p==null)
      return "";
    return p.getInfo().toString()+"\n"+visita(p.getLink());
  }

  public String elenco() {
    return visita(head);
  }

  public String toString() {
    Nodo p = head;
    String lista = new String("head->");

    if (p==null)
      return lista+"null";
```

```
    while (p!= null) {
      lista = lista+"["+p.getInfo().toString()+"|";
        if (p.getLink()==null)
          lista = lista+"null]";
        else
          lista = lista+"+]->";
          p = p.getLink();
    }
    return lista;
  }

  public static void main (String[] args) {
    Invitato i1 = new Invitato("Bianchi Giovanni",'M',"0586 854822");
    Invitato i2 = new Invitato("Rossi Marta",'F',"0586 844853");
    Invitato i3 = new Invitato("Neri Marco",'M',"0586 444722");
    Invitato i4 = new Invitato("Verdi Roberta",'F',"0586 974824");
    Festa f = new Festa();

    f.inserisciInTesta(i1);
    f.inserisciInTesta(i2);
    f.inserisciInCoda(i3);
    try {f.inserisciInPosizione(i4, 2);}
    catch (FestaException exception) {
      System.out.println(exception.getError());
    }

    System.out.println("Visita ricorsiva: ");
    System.out.println(f.elenco());
    System.out.println("------------------");

    System.out.println(f.toString());
    System.out.println("------------------");
    try {f.eliminaInPosizione(2);}
    catch (FestaException exception) {
      System.out.println(exception.getError());
    }
    System.out.println(f.toString());
    System.out.println("------------------");
    try {f.eliminaInCoda();}
    catch (FestaException exception) {
      System.out.println(exception.getError());
    }
    System.out.println(f.toString());
    System.out.println("------------------");
    try {f.eliminaInTesta();}
    catch (FestaException exception) {
      System.out.println(exception.getError());
    }
    System.out.println(f.toString());
    System.out.println("------------------");
    try {f.eliminaInPosizione(5);}
    catch (FestaException exception) {
      System.out.println(exception.getError());
    }
    System.out.println(f.toString());
    System.out.println("------------------");
```

```
      try {f.eliminaInPosizione(1);}
      catch (FestaException exception) {
        System.out.println(exception.getError());
      }
      System.out.println(f.toString());
      System.out.println("------------------");
      try {f.eliminaInPosizione(1);}
      catch (FestaException exception) {
        System.out.println(exception.getError());
      }
    }
}
```

Il metodo *main* della classe *Festa* ne effettua il test dei metodi e produce il seguente output:

```
Visita ricorsiva:
Rossi Marta F 0586 844853
Verdi Roberta F 0586 974824
Bianchi Giovanni M 0586 854822
Neri Marco M 0586 444722
------------------
head->[Rossi Marta F 0586 844853|+]->[Verdi Roberta F 0586 974824|+]->[Bianchi Giovanni
M 0586 854822|+]->[Neri Marco M 0586 444722|null]
------------------
head->[Rossi Marta F 0586 844853|+]->[Bianchi Giovanni M 0586 854822|+]->[Neri Marco M
0586 444722|null]
------------------
head->[Rossi Marta F 0586 844853|+]->[Bianchi Giovanni M 0586 854822|null]
------------------
head->[Bianchi Giovanni M 0586 854822|null]
------------------
Posizione 5 errata
head->[Bianchi Giovanni M 0586 854822|null]
------------------
head-> null
------------------
Lista vuota
```

I metodi della classe *Festa* che accettano come parametri, o restituiscono come risultato, oggetti di classe *Nodo* sono definiti `private`: infatti l'esposizione di un oggetto di classe *Nodo* ne esporrebbe anche il relativo *link* e, di conseguenza, la possibilità di manipolare esternamente dalla classe gli elementi della lista, violando il principio di incapsulazione.

1.2 Liste multiple

Le liste del tipo trattato fino a questo punto sono definite «semplici». Facendo ancora riferimento all'esempio della festa presentato in apertura di capitolo, se si accetta il fatto che ogni invitato possa invitare a sua volta altre persone, una lista semplice non è più sufficiente per rappresentare adeguatamente la situazione.

Una possibile soluzione è quella di utilizzare nodi che, invece di avere un solo riferimento per collegare ogni invitato al successivo, abbiano anche un riferimento alla lista dei propri invitati. Una struttura che consente di gestire situazioni di questo tipo è denominata **lista multipla**. Schematicamente una lista multipla può essere rappresentata come in FIGURA 4, dove:

- per semplicità la componente informativa (*info*) è limitata a un'etichetta identificativa invece di riportare il riferimento a un oggetto di classe *Invitato* come dovrebbe essere;
- *sub* rappresenta per ogni nodo il riferimento all'eventuale sottolista di invitati da parte di uno specifico invitato;
- *link* rappresenta per ogni nodo il riferimento all'eventuale invitato successivo dello stesso «livello»: invitati direttamente da chi organizza la festa o una qualsiasi sottolista di invitati dalla stessa persona.

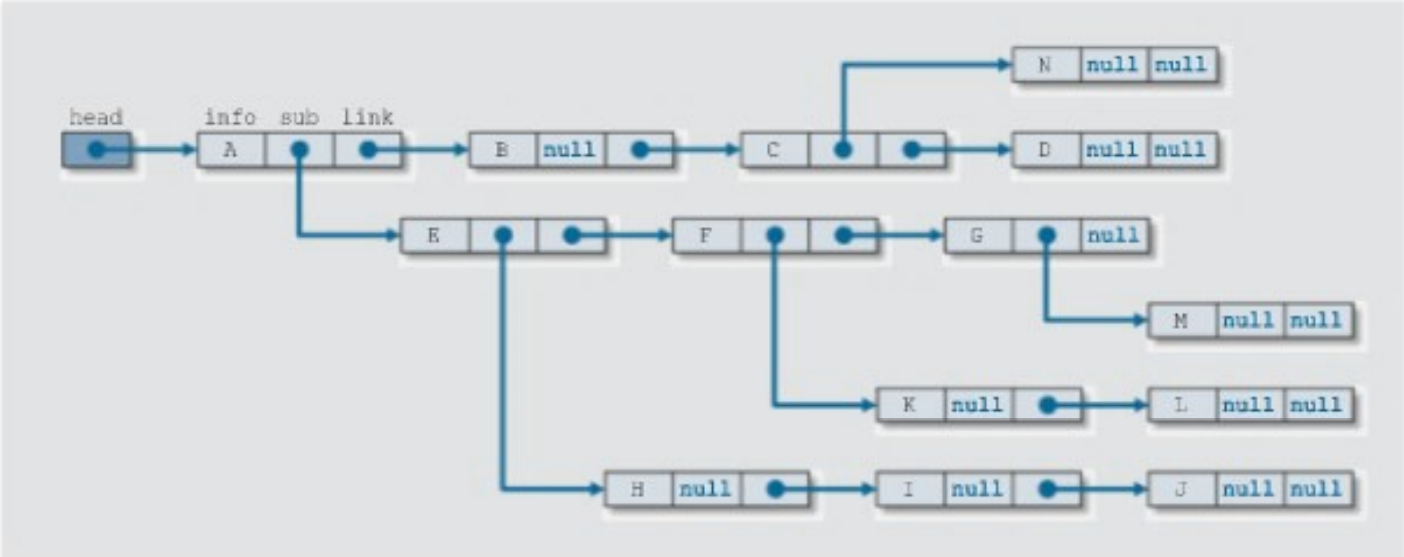


FIGURA 4

Volendo visitare una lista multipla è necessario stabilire un criterio di visita. Per esempio si potrebbe stabilire di esaminare un nodo e tutte le sue eventuali sottoliste prima di passare all'elemento riferito dal *link* del nodo stesso. Applicando tale criterio, la sequenza dei nodi visitati è *A*, [*E*, [*H*, *I*, *J*], *F*, [*K*, *L*], *G*, [*M*]], *B*, *C*, [*N*], *D*: le parentesi servono a evidenziare la gerarchia dei vari gruppi di invitati.
È evidente che per eseguire questo tipo di visita sarà necessario memorizzare, ogni volta che si scende di un livello, il riferimento al nodo corrente per riprendere successivamente dalla corretta posizione la scansione della lista. Come vedremo, una pila si presta molto bene per questo tipo di operazione, in quanto l'ultimo riferimento memorizzato sarà il primo a essere recuperato per poi risalire di un livello e verificare se vi sono ancora elementi disponibili per riprendere la scansione in avanti. Il verificarsi della doppia condizione di *link* con valore `null` e di pila vuota indica il termine dell'algoritmo di visita.

2 Il *pattern* di progettazione *Iterator*

I *design pattern* sono soluzioni generiche di progettazione software applicabili a problemi ricorrenti: questo approccio si è diffuso a partire dalla pubblicazione nel 1995 di *Design patterns: elementi per il riuso di software ad oggetti* di Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, da allora noti come la «Banda dei quattro» (*Gang of Four* o GoF). Di seguito viene brevemente descritto il *design pattern* ***Iterator*** e una sua possibile applicazione alla lista.

In generale, dato un oggetto aggregato (cioè un oggetto che contiene altri oggetti per fornire di questi una visione unitaria, come per esempio una lista), un **Iteratore** (***Iterator***) è un oggetto che rende disponibili metodi per accedere sequenzialmente ai singoli elementi dell'oggetto aggregato senza esporne la rappresentazione interna.

ESEMPIO

Da un punto di vista logico il selettore dei canali di un televisore è schematizzabile come un iteratore: piuttosto che disporre di un pulsante (o di una combinazione di essi) per selezionare ogni canale sintonizzabile, è possibile utilizzare i pulsanti *Successivo* e *Precedente* per scorrere sequenzialmente i canali. In questo modo l'utente non ha la necessità di conoscere la posizione esatta di un canale per poterlo individuare. In questo esempio gli elementi in gioco sono:

- il selettore dei canali che corrisponde all'iteratore;
- i pulsanti *Successivo* e *Precedente* del telecomando che corrispondono ai metodi resi disponibili dall'iteratore;
- l'insieme dei canali che corrispondono all'aggregato degli elementi.

Come conseguenza della disponibilità di un iteratore lo scorrimento dei canali può avvenire in ordine discendente o ascendente a partire da un qualsiasi elemento dell'aggregato (canale attualmente selezionato): l'iteratore semplifica l'interfaccia verso l'aggregato consentendo la selezione dei soli canali disponibili.

2.1 Iteratore e lista

Le motivazioni per l'applicazione di un iteratore a una lista sono le seguenti:

- permettere l'accesso agli elementi della lista senza esporre la sua struttura interna;
- fare in modo che l'accesso agli elementi della lista avvenga mediante metodi che non sono direttamente parte dell'interfaccia della classe che implementa la lista stessa, ma di una classe diversa – l'iteratore – i cui oggetti sono restituiti dall'invocazione di uno specifico metodo.

Il diagramma UML che schematizza la soluzione è mostrano in FIGURA **5**, dove *Tipo* è la classe della componente informatica dei nodi della lista e la dipendenza della classe *Lista* dalla classe *Iteratore* esplicita il fatto che il metodo *getIterator* restituisce un oggetto di classe *Iteratore*. I metodi dell'iteratore sono denominati secondo la tradizione degli iteratori del linguag-

gio di programmazione Java: *hasNext* restituisce `true` se è possibile avanzare nella sequenza di elementi della lista, `false` altrimenti; *next* restituisce l'elemento successivo della lista rispetto all'ultimo fornito.

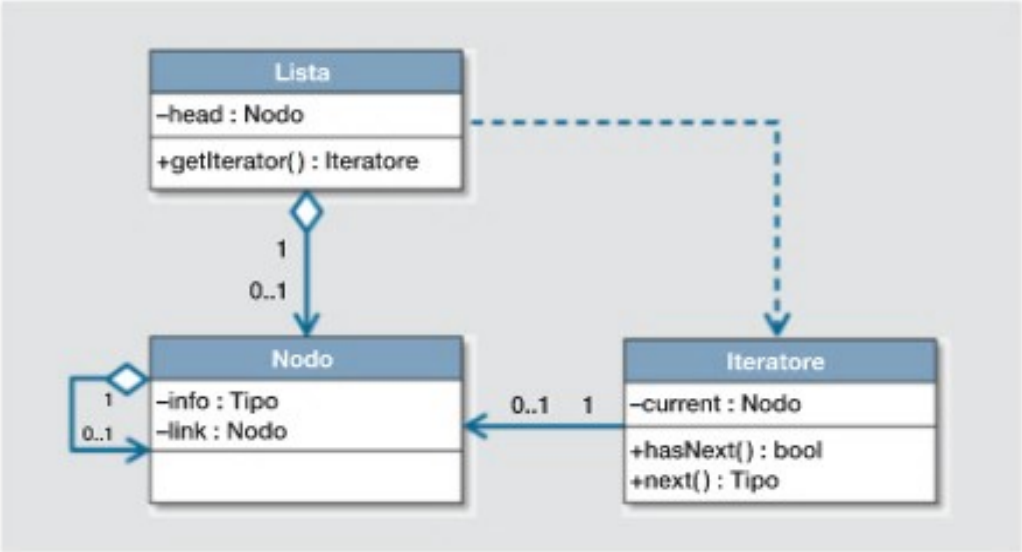


FIGURA 5

ESEMPIO

Ritornando all'esempio della lista degli invitati alla festa, una semplice classe *Iteratore* potrebbe essere così implementata:

```
public class Iteratore {
   private Nodo nodo;

   public Iteratore(Nodo nodo) {
      this.nodo = nodo;
   }

   public boolean hasNext() {
      return (!(nodo==null));
   }

   public Invitato next() {
      if (nodo==null)
         return null;
      Invitato persona = new Invitato(nodo.getInfo());
      nodo = nodo.getLink(); // avanzamento di posizione nella lista
      return persona;
   }
}
```

Nella classe *Festa* il metodo *getIterator* è di semplice codifica:

```
public Iteratore getIterator() {
   Iteratore i = new Iteratore(head);
   return i;
}
```

Il relativo test nel metodo *main* potrebbe essere il seguente

```
...
Iteratore iteratore = f.getIterator();
Invitato invitato;
```

```
while (iteratore.hasNext()) {
  invitato = iteratore.next();
  System.out.println(invitato);
}
...
```

e produce un output come quello che segue:

```
Rossi Marta F 0586 844853
Verdi Roberta F 0586 974824
Bianchi Giovanni M 0586 854822
Neri Marco M 0586 444722
```

3 Le classi *nested*

Le classi *nested* sono *classi nidificate* o interne, ovvero classi contenute in altre classi. Anche se spesso vengono usati i termini *nested* e *inner* in maniera intercambiabile le *nested classes* sono divise in due categorie fondamentali: *static* e *non-static*: le prime sono chiamate semplicemente *static nested classes*, le seconde sono dette *inner classes*.

Nel dettaglio le classi *nested* previste da Java sono articolate in:

- *static class*: dichiarate come membro statico di un'altra classe;
- *inner class*: dichiarate come membro non statico di un'altra classe;
- *local inner class*: dichiarate all'interno del metodo di un'altra classe;
- *anonymous inner class*: sono simili alle precedenti ma scritte come un'espressione che ritorna un oggetto.

I motivi che possono indurre all'uso di classi nidificate sono:

- *organizzazione logica*: talvolta può essere ragionevole definire una classe all'interno di un'altra specie quando la prima è funzionale alla seconda e non viene utilizzata in altri contesti;
- *migliorare l'incapsulamento*: le classi nidificate sono facilitate nell'accedere agli attributi delle classi che le contengono (date due classi, *A* e *B*, nidificando la classe *B* nella classe *A* è possibile permettere a *B* l'accesso ai membri di *A* anche se questi sono dichiarati privati e al tempo stesso rendere *B* invisibile all'esterno di *A*);
- *migliorare la leggibilità e la manutenzione del codice*: nidificare piccole classi in altre classi permette la collocazione del codice delle prime «vicino» a dove questo viene usato; in tal modo si evita tra l'altro di dover creare un file distinto per ogni tipo di classe.

Anche se il grado di nidificazione delle classi non ha limite (per esempio la classe *A* può contenere la classe *B* che contiene la classe *C* che contiene la classe *D*...), generalmente, a meno di una progettazione poco appropriata, è piuttosto raro che si abbia la necessità di andare oltre un livello di nidificazione.

Static class

Le classi di questo tipo sono le più semplici e sono dichiarate come membro statico di un'altra classe. Per esempio:

```
public class ClasseEsterna {
  ...
  public static class ClasseInterna {
    ...
  }
}
```

Una classe di tipo *static*, come qualsiasi membro statico di una classe, è accedibile usando il nome della classe esterna che la contiene. Nel caso dell'esempio appena visto:

```
ClasseEsterna.ClasseInterna
```

Sempre relativamente all'esempio di prima, per l'istanziamento di un oggetto da *ClasseInterna*, si può usare:

```
ClasseEsterna.ClasseInterna oggettoInterno = new ClasseEsterna.ClasseInterna();
```

Di fatto, l'unica situazione che, in qualche modo, giustifichi la creazione di una classe statica nidificata è quando essa debba poter avere accesso ai membri statici privati della classe che la contiene.

Inner class

Un oggetto istanza di una classe *inner* esisterà solo nell'ambito di una istanza della classe contenitore.

Nel seguente esempio *ClasseInterna* è dichiarata come membro non statico di *ClasseEsterna* e pertanto un'istanza di *ClasseInterna* ha l'accesso diretto ai metodi e agli attributi dell'istanza di *ClasseEsterna* che la contiene:

```
public class ClasseEsterna {
  ...
  public class ClasseInterna {
    ...
  }
  private void metodoA() {
    ClasseInterna ci = new ClasseInterna();
  }
}
```

Per istanziare un oggetto di una classe interna, è necessario istanziare prima un oggetto della classe contenitore come nell'esempio seguente:

```
ClasseEsterna alfa = new ClasseEsterna();
ClasseEsterna.ClasseInterna beta = alfa.new ClasseInterna();
```

Relativamente all'esempio della gestione degli invitati a una festa visto nei precedenti paragrafi, la struttura del codice potrebbe essere rivista secondo lo schema riportato di seguito:

```
public class Festa {
  ... attributi e metodi della classe Festa ...

    private class Nodo {
    ... attributi e metodi della classe Nodo ...
    }

    public class Iteratore {
    ... attributi e metodi della classe Iteratore ...

    }

    public class FestaException extends Exception {
     ... attributi e metodi della classe FestaException ...
    }
}
```

Questo tipo di soluzione presenta le seguenti caratteristiche:

- il codice delle singole classi non è stato minimamente modificato rispetto a quanto visto nei paragrafi precedenti;
- tutto il codice è memorizzato in un unico file (*Festa.java*);
- la classe visibile è *Festa* che espone *Iteratore* e *FestaException*, mentre la classe *Nodo* è una classe interna che, essendo stata dichiarata *private*, è utilizzabile solo da *Festa* che la incapsula e la nasconde all'esterno.

Local inner class

Una classe *local inner* è una classe dichiarata nel corpo di un metodo e solo all'interno di esso è disponibile per essere operativa (istanziare oggetti e rendere disponibile i suoi metodi). Una classe di questo tipo può accedere alle variabili locali della classe esterna e a quelle del metodo che la ospita. L'esempio che segue produce in output i valori 10 e 20:

```
public class ClasseEsterna {
  private int x = 10;

  public void creaClasseLocalInner() {
    int y = 20;

    class LocalInner {
      public void accediEsterna() {
        System.out.println(x);
        System.out.println(y);
      }
    }
    LocalInner li = new LocalInner();
    li.accediEsterna();
  }
}

public class DemoLocalInner {
  public static void main(String[] args) {
    ClasseEsterna oggettoE = new ClasseEsterna();
    oggettoE.creaClasseLocalInner();
  }
}
```

Anonymous inner class

Una classe *anonymous inner* costituisce una particolarità sintattica per scrivere una classe *local inner* senza necessità di assegnarle un nome esplicito.

L'esempio che segue visualizza in output la frase «Gatto Anonimo»:

```
public class DemoAnonymousInner {
  public static void main(String[] args) {
    Gatto gatto = new Gatto() {
      public void unGatto () {
        System.out.println("Gatto Anonimo");
      }
    }; // fine corpo classe anonymous
    // gatto contiene un oggetto della sottoclasse
    // anonymous di Gatto
    gatto.unGatto();
  }
}

public class Gatto {
  public void unGatto() {
    System.out.println("Gatto standard");
  }
}
```

Nel codice, la linea `Gatto gatto = new Gatto() {`, termina con una graffa e non un punto e virgola. Questa graffa apre la definizione di classe e dichiara una nuova classe senza nome (*anonymous*). Nel corpo della nuova sottoclasse di Gatto il metodo *unGatto()* viene sovrascritto. Questo è il punto cruciale nella definizione di una classe interna anonima perché vengono ignorati uno o più metodi della superclasse.

4 La pila e la coda

Due strutture dati molto usate nella pratica sono la **pila** (*stack*) e la **coda** (*queue*).

4.1 Pila

La **pila** è una struttura dati che adotta una politica di tipo LIFO (*Last-In First-Out*), ovvero l'ultimo elemento inserito è sempre il primo a essere estratto.

ESEMPIO Un binario morto dove sono parcheggiati temporaneamente dei vagoni è una pila: l'ultimo vagone a essere inserito sarà necessariamente il primo a essere estratto.

La pila è, relativamente al suo contenuto, caratterizzata da due operazioni fondamentali: **inserimento** (***push***) ed **estrazione** (***pop***). Schematicamente una pila può essere rappresentata come in FIGURA **6**, dove sono riportate le operazioni di inserimento e di estrazione di un elemento.

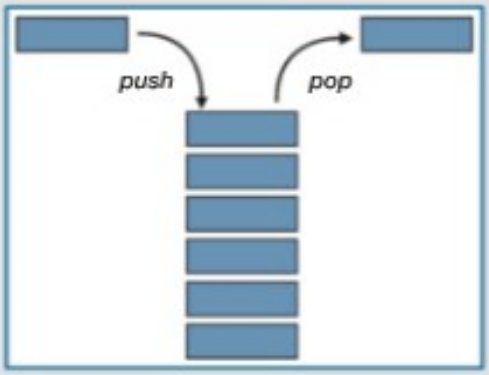


FIGURA 6

Dal punto di vista della sua implementazione questo tipo di struttura dati può essere facilmente realizzata impiegando una lista. Questa soluzione è efficiente perché garantisce sia la dinamicità del numero di elementi contenuti nella coda sia una corretta gestione dell'occupazione di memoria. In questo scenario le operazioni *push* e *pop* sono implementate banalmente con i metodi visti in precedenza per l'inserimento e l'estrazione in testa degli elementi di una lista.

ESEMPIO

Vediamo un esempio di realizzazione di una pila in linguaggio Java dove sono riutilizzate le classi *Nodo* e *Invitato* definite negli esempi precedenti:

```
public class Pila {
  private Nodo head;

  public Pila() {
    head = null;
  }

  public Nodo creaNodo(Invitato persona, Nodo link) {
    Nodo p = new Nodo(persona);

    p.setLink(link);
    return p;
  }
  public void push(Invitato persona) {
    Nodo p;
    // inserimento in testa
    p = creaNodo(persona, head);
    head = p;
  }

  public Invitato pop() {
    Nodo p;

    if (head==null) // pila vuota?
      return null;
    p = head;
    head = head.getLink();
    return p.getInfo;
  }
```

```
  public String toString() {
    Nodo p = head;
    String lista = new String("head->");

    if (p==null)
      return lista+"null";
    while (p != null) {
      lista = lista+"["+p.getInfo().toString()+"|";
        if (p.getLink()==null)
          lista = lista+"null]";
        else
          lista = lista+"+]->";
          p = p.getLink();
    }
    return lista;
  }

  public static void main (String[] args) {
    Invitato inv1 = new Invitato("Bianchi Giovanni",'M',"0586 854822");
    Invitato inv2 = new Invitato("Rossi Marta",'F',"0586 844853");
    Pila pila = new Pila();
    pila.push(inv1);
    pila.push(inv2);
    System.out.println(pila.toString());

    Invitato inv = pila.pop();
    if (inv==null)
      System.out.println("Pila vuota");
    else
      System.out.println(inv);
    System.out.println(pila.toString());
    inv = pila.pop();
    if (inv==null)
      System.out.println("Pila vuota");
    else
      System.out.println(inv);
    System.out.println(pila.toString());
    inv = pila.pop();
    if (inv==null)
      System.out.println("Pila vuota");
    else
      System.out.println(inv);
  }
}
```

Il metodo *main* usato per testare i metodi della classe *Pila* produce il seguente output:

```
head->[Rossi Marta F 0586 844853|+]->[Bianchi Giovanni M 0586 854822|null]
Rossi Marta F 0586 844853
head->[Bianchi Giovanni M 0586 854822|null]
Bianchi Giovanni M 0586 854822
head->null
Pila vuota
```

Il tipo di formattazione utilizzato per l'output è servito solo a mettere in evidenza la struttura a lista utilizzata.

4.2 Coda

La **coda** è una struttura dati che adotta una politica di tipo FIFO (*First-In First-Out*), ovvero il primo elemento a essere inserito è sempre il primo a essere estratto.

ESEMPIO Una fila di persone ben educate alla cassa del supermercato costituiscono una coda.

La coda è una struttura dati utilizzata spesso per memorizzare temporaneamente informazioni relative a persone, oggetti o eventi che devono essere elaborate in un momento successivo; in questo modo la coda realizza la funzione di un *buffer*.

Come nel caso della pila ci sono due operazioni fondamentali: l'**inserimento** (***enqueue***) e l'**estrazione** (***dequeue***). Schematicamente una coda può essere rappresentata come in FIGURA 7, dove sono riportate le operazioni di inserimento e di estrazione di un elemento.

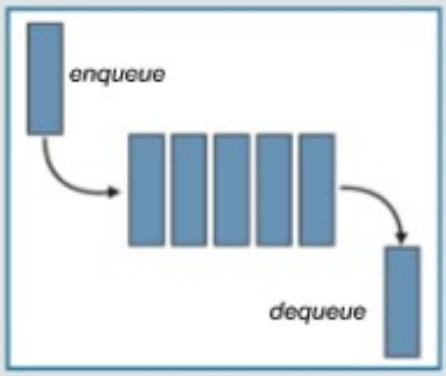


FIGURA 7

Come la pila, anche la coda può essere memorizzata mediante una lista: in questo caso le due operazioni sono implementate dai metodi per l'inserimento in coda e l'estrazione in testa degli elementi.

Dal momento che implementando una coda con una lista è necessario fare riferimento sia all'elemento di testa, per l'estrazione, sia a quello di coda, per l'inserimento, è ragionevole definire e gestire due diversi attributi di riferimento: *head* per l'accesso al primo elemento della lista e *tail* per l'accesso all'ultimo elemento della lista.
In questo modo è possibile evitare lo scorrimento della lista per ogni operazione di inserimento.

ESEMPIO Vediamo un esempio di realizzazione di una pila in linguaggio Java dove sono riutilizzate le classi *Nodo* e *Invitato* definite negli esempi precedenti:

```
public class Coda {
    private Nodo head;
    private Nodo tail;
    public Coda() {
```

```
    head = null;
    tail = null;
  }
  private Nodo creaNodo(Invitato persona, Nodo link) {
    Nodo p = new Nodo(persona);
    p.setLink(link);
    return p;
  }
  public void enqueue(Invitato persona) {
    Nodo p = creaNodo(persona, null);
    if (head==null) { // coda vuota?
      tail=p;
      head=tail;
    }
    else {
      tail.setLink(p);
      tail = p;
    }
  }
  public Invitato dequeue() {
    Nodo p;
    if (head==null) // coda vuota?
      return null;
    p = head;
    head = head.getLink();
    if (head==null) // ultimo elemento della coda?
      tail=null;
    return p.getInfo();
  }
  public String toString() {
    Nodo p = head;
    String lista;

    if (p==tail) // coda di un solo elemento?
      lista = new String("head/tail->");
    else
      lista = new String("head->");
    if (p==null)
      return lista + "null";
    while (p != null) {
      lista = lista+"[" + p.getInfo().toString() + "|";
      if (p.getLink()==tail)
        lista = lista + "+]-(tail)->";
      else {
        if (p.getLink()==null)
          lista = lista + "null]";
        else
          lista = lista + "+]->";
      }
        p = p.getLink();
    }
    return lista;
  }
  public static void main (String[] args) {
    Invitato inv1 = new Invitato("Bianchi Giovanni",'M',"0586 854822");
```

```
    Invitato inv2 = new Invitato("Rossi Marta",'F',"0586 844853");
    Coda coda = new Coda();

    coda.enqueue(inv1);
    coda.enqueue(inv2);
    System.out.println(coda.toString());
    Invitato inv = coda.dequeue();
    if (inv==null)
      System.out.println("Coda vuota");
    else
      System.out.println(inv);
    System.out.println(coda.toString());
    inv = coda.dequeue();
    if (inv==null)
      System.out.println("Coda vuota");
    else
      System.out.println(inv);
    System.out.println(coda.toString());
    inv = coda.dequeue();
    if (inv==null)
      System.out.println("Coda vuota");
    else
      System.out.println(inv);
  }
}
```

Il metodo *main* usato per testare i metodi della classe *Pila* produce il seguente output:

```
head->[Bianchi Giovanni M 0586 854822|+]-(tail)->[Rossi Marta F 0586 844853|null]
Bianchi Giovanni M 0586 854822
head/tail->[Rossi Marta F 0586 844853|null]
Rossi Marta F 0586 844853
head/tail->null
Coda vuota
```

5 Alberi

Un **albero** è una struttura gerarchica in cui ogni elemento (detto nodo), eccetto uno, ha un solo elemento «padre» e tutti gli elementi possono avere elementi «figli». L'unico elemento privo di padre è la «radice» dell'albero, mentre gli elementi privi di figli sono le «foglie».

Gli alberi sono una struttura dati molto utilizzata in informatica perché permettono di memorizzare informazioni in modo dinamico, efficiente e ordinato.

I software relativi a situazioni in cui sia necessario esplorare uno spazio di soluzioni alternative per un dato problema (per esempio l'intelligenza artificiale), ricorrono spesso alla rappresentazione interna delle possibilità di scelta in forma di albero. In particolare, nel caso di giochi, ogni nodo di un livello dell'albero rappresenta una diversa possibile scelta rispetto alla situazione dei nodi di livello superiore. Il *backtracking*, per esempio, è una tecnica per trovare soluzioni a problemi in cui devono essere soddisfatti dei vincoli enumerando tutte le possibili soluzioni e scartando quelle che non li soddisfano (per esempio il gioco degli scacchi).

ESEMPIO

Il sistema dell'organizzazione del disco in file e directory è normalmente reso disponibile all'utente di un computer da parte del sistema operativo nella forma di un albero di cui i file rappresentano le foglie e la directory *root* la radice (FIGURA **8**).

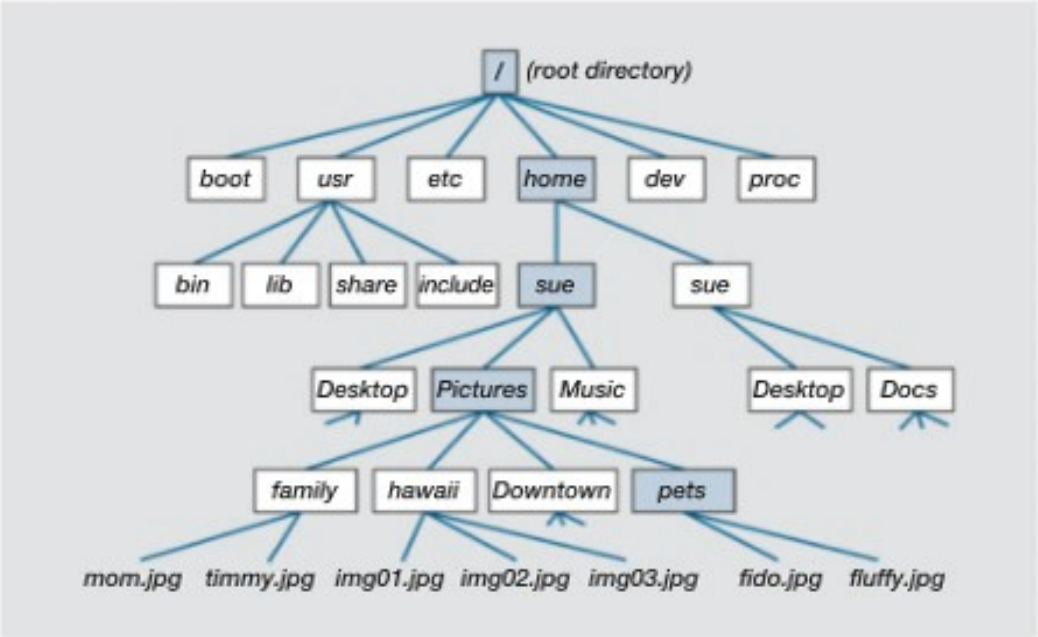


FIGURA **8**

5.1 Alberi generici

Un albero è un insieme non vuoto di **nodi** (elementi aventi valenza informativa) e di **archi** (elementi aventi valenza connettiva che realizzano i collegamenti tra coppie di nodi), tale che sia:

- **connesso**, cioè presi due nodi qualsiasi esista un percorso semplice tra di essi;
- **aciclico**, ovvero non esista alcun percorso semplice in cui il nodo iniziale e quello finale siano coincidenti.

Per «percorso semplice» si intende una sequenza di coppie di nodi *distinti* collegate da un arco che rappresenta un cammino sugli archi dell'albero.

ESEMPIO

La FIGURA **9** rappresenta un albero in cui è evidenziato il percorso semplice (*d*, *a*), (*a*, *b*), (*b*, *g*) dal nodo *d* al nodo *g* passando per i nodi *a* e *b*.

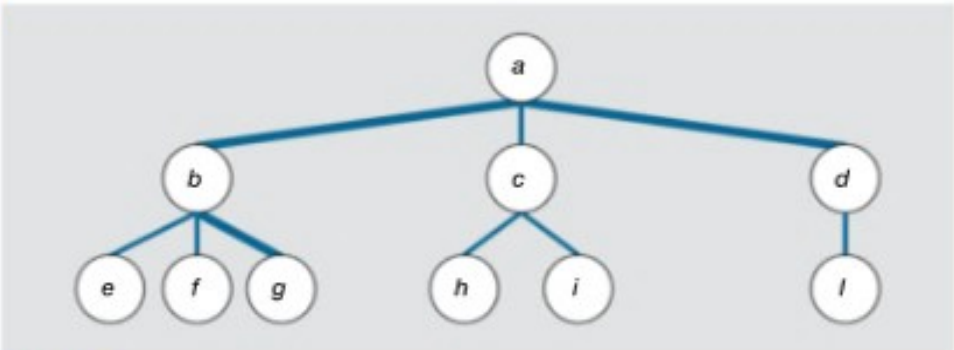


FIGURA **9**

In un albero di n nodi si hanno le seguenti proprietà:

- ha $n - 1$ archi;
- esiste un solo percorso semplice tra ogni coppia di nodi.

Un albero è definito **albero con radice** se uno dei nodi dell'albero viene scelto come nodo radice.

Nel caso di albero con radice, se un nodo x compare in almeno un percorso semplice che parte dal nodo radice successivamente al nodo y, allora x è «discendente» di y e, se x e y sono direttamente connessi da un arco, allora x è **figlio** di y. Viceversa, se un nodo x compare in almeno un percorso semplice che arriva al nodo radice successivamente al nodo y, allora x è «ascendente» di y e, se x e y sono direttamente connessi da un arco, allora x è **padre** di y. I nodi che non hanno figli sono denominati **foglie**.

In un albero con radice, se il nodo x è figlio del nodo y, allora il nodo y è padre del nodo x.

ESEMPIO

Nell'albero di FIGURA **10**, scelto *a* come nodo radice, i nodi *b*, *c* e *d* sono figli di *a*; i nodi *e*, *f* e *g* sono figli di *b*; i nodi *h* e *i* sono figli di *c*; il nodo *l* è figlio di *d*.

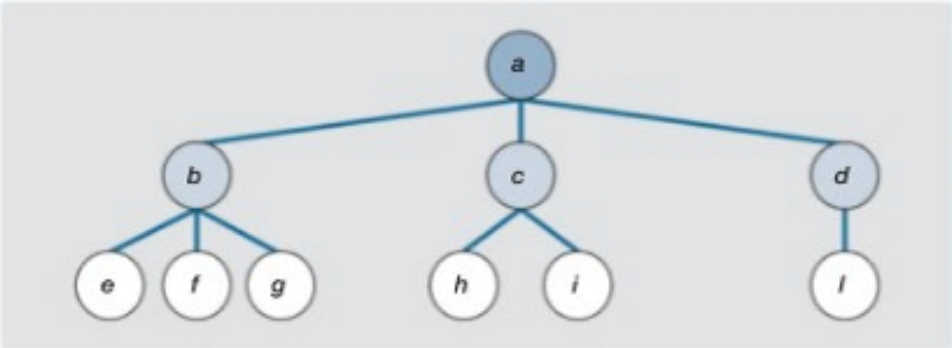


FIGURA **10**

Viceversa, il nodo *b* è padre dei nodi *e*, *f* e *g*; il nodo *c* è padre dei nodi *h* e *i*; il nodo *d* è padre del nodo *l*; il nodo *a* è padre dei nodi *b*, *c* e *d*.

In un albero con radice i nodi figli dello stesso nodo sono definiti **fratelli**. L'insieme dei fratelli con il padre è detto **famiglia**.

ESEMPIO

Nell'albero rappresentato nell'esempio precedente i nodi *b*, *c* e *d* sono fratelli, così come i nodi *e*, *f* e *g* e i nodi *h* e *i*. I nodi *e*, *f*, *g*, *h*, *i* ed *l* sono le foglie dell'albero.

In un albero con radice:

- il nodo radice non ha padre; ogni altro nodo dell'albero può avere più figli, ma un solo padre; eliminando il nodo radice si ottiene un albero (**sottoalbero**) per ognuno dei nodi figli;

- la **profondità** di un nodo è costituita dal numero di archi che sono presenti sul percorso semplice che congiunge il nodo stesso con il nodo radice (la radice ha profondità 0);
- i nodi possono essere classificati in **livelli**: ogni nodo appartiene al livello coincidente con la propria profondità.

ESEMPIO Nell'albero di FIGURA **11**, scelto *a* come nodo radice, i nodi *b*, *c* e *d* sono i nodi radice dei rispettivi sottoalberi. La profondità dei nodi *b*, *c* e *d* è 1 (*b*, *c* e *d* appartengono al livello 1 dell'albero), mentre la profondità dei nodi *e*, *f*, *g*, *h*, *i* ed *l* è 2 (e, *f*, *g*, *h*, *i* ed *l* appartengono al livello 2 dell'albero); di conseguenza la profondità massima dell'albero è 2.

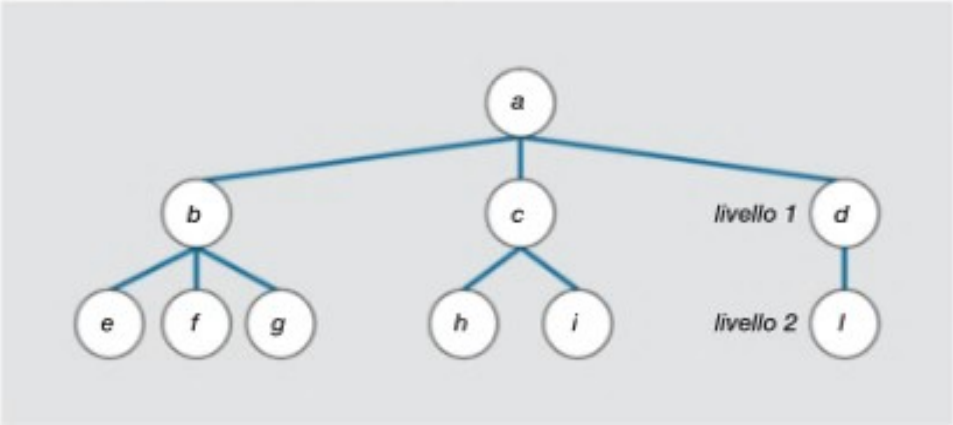


FIGURA **11**

Un albero con radice può essere memorizzato ricorrendo a una lista multipla come quella di FIGURA **12**, che rappresenta l'albero dell'esempio precedente, dove:

- *ptr* è il riferimento alla radice dell'albero;
- ogni nodo dell'albero ha tre componenti: *info* (componente informativa), *pfc* (riferimento al primo figlio) e *pfs* (riferimento al primo fratello).

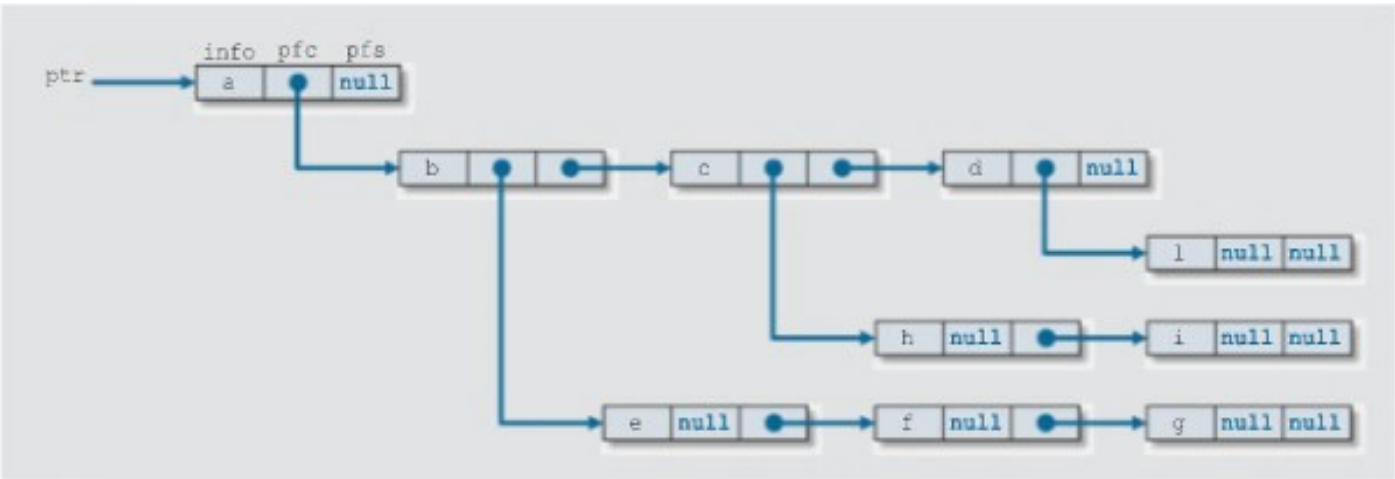


FIGURA **12**

OSSERVAZIONE L'acronimo pfc sta per *pointer first child*, mentre l'acronimo pfs sta per *pointer first sibling*.

La memorizzazione di un albero mediante una lista multipla impone un ordinamento dei figli di ogni nodo in base al quale è possibile parlare di «primo figlio», «secondo figlio», ... e di conseguenza di «primo sottoalbero», «secondo sottoalbero», ...

ESEMPIO

Tralasciando la definizione delle operazioni il diagramma UML delle classi che consentono di rappresentare un albero implementato in forma di lista multipla è quello di FIGURA **13**.

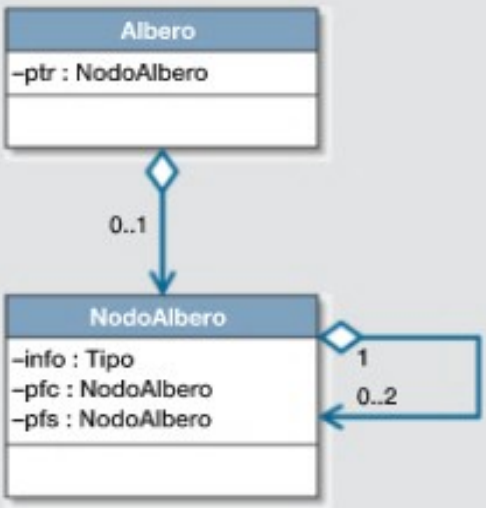


FIGURA **13**

Nel linguaggio di programmazione Java la classe *NodoAlbero* è di immediata realizzazione (è stato lasciato non definito il tipo della componente informativa):

```
public class NodoAlbero {

  private ... info;
  private NodoAlbero pfc;
  private NodoAlbero pfs;

  public NodoAlbero(... info) {
    this.info = new ...(info);
    pfc = null;
    pfs = null;
  }

  public void setInfo(... info) {
    this.info = new ...(info);
  }

  public ... getInfo() {
    return new ...(info);
  }

  public void setFirstChild(NodoAlbero pfc) {
    this.pfc = pfc;
  }

  public NodoAlbero getFirstChild() {
    return pfc;
  }
```

```
    public void setFirstSibling(NodoAlbero pfs) {
      this.pfs = pfs;
    }

    public NodoAlbero getFirstSibling() {
      return pfs;
    }
  }
```

La classe che rappresenta l'albero avrà come attributo il riferimento al nodo radice:

```
public class Albero {
  private NodoAlbero ptr;
  ...
  ...
  ...
}
```

Per visitare tutti i nodi di un albero memorizzato come lista multipla è necessario utilizzare un criterio di visita che in pratica definisce un ordinamento dei nodi. Le due classiche tipologie di visita dei nodi di un albero sono la **visita in ordine anticipato** e la **visita in ordine differito**.

Algoritmo di visita in ordine anticipato

0) Visita la radice. Se $n > 0$ è il numero di sottoalberi della radice:
1) Visita il primo sottoalbero in ordine anticipato;
2) Visita il secondo sottoalbero in ordine anticipato;
…;
n) Visita l'n-esimo sottoalbero in ordine anticipato.

L'algoritmo di visita dell'albero è definito ricorsivamente; di conseguenza è naturale che anche l'implementazione nel linguaggio di programmazione Java sia un metodo ricorsivo

```
private void visitaAnticipata(NodoAlbero p) {
  if (p == null)
    return;
  esamina(p.getInfo());
  if (p.getFirstChild() != null)
    visitaAnticipata(p.getFirstChild());
  if (p.getFirstSibling() != null)
    visitaAnticipata(p.getFirstSibling());
}
```

dove il metodo *esamina* rappresenta un metodo generico di elaborazione applicato alla componente informativa dell'elemento riferito da *p* (per esempio la visualizzazione del suo contenuto). Il metodo ricorsivo privato *visitaAnticipata* può essere invocato da un metodo pubblico in *overloading* a partire dal nodo radice dell'albero:

```
public void visitaAnticipata() {
  visitaAnticipata(ptr);
}
```

ESEMPIO

Visitando l'albero di FIGURA **14** in modo anticipato si ottiene la sequenza *a, b, e, f, g, c, h, i, d, l*.

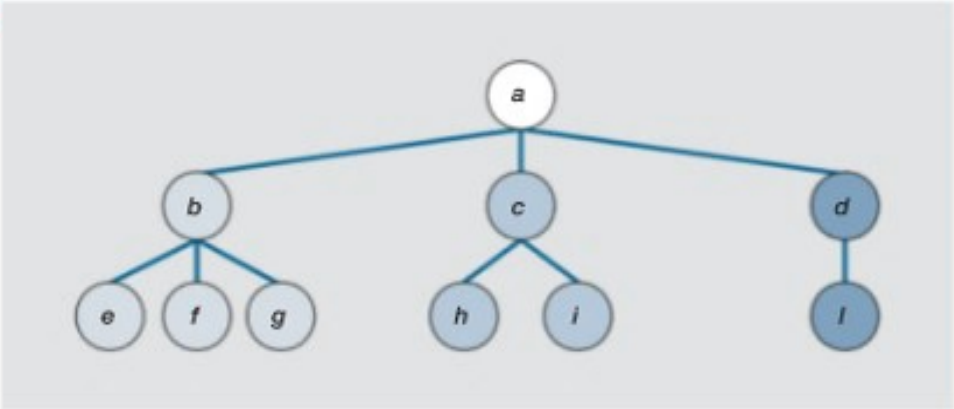


FIGURA **14**

Una formalizzazione non ricorsiva dell'algoritmo di visita in ordine anticipato richiede l'impiego di una struttura dati per la memorizzazione temporanea dei riferimenti dei nodi. L'algoritmo per ogni nodo visita in primo luogo, se non è vuota, la sottolista dei nodi figli e successivamente gli eventuali nodi fratelli. Per come è memorizzato l'albero, questo implica che prima di scorrere la sottolista di un nodo figlio è necessario memorizzare il riferimento al nodo corrente, per poter riprendere in seguito da quel nodo la visita dei nodi fratelli. Dato che, quando per un nodo non si hanno ulteriori nodi fratelli, è necessario riprendere la visita dal nodo da cui era iniziata la scansione dell'ultima sottolista visitata in ordine di tempo, una soluzione adeguata è quella di utilizzare una pila nel seguente modo:

- prima di visitare la sottolista di un nodo figlio si memorizza il riferimento al nodo corrente;
- se un nodo non ha figli, si passa all'esame del nodo fratello da lui direttamente riferito; se non vi sono né nodi figli né nodi fratelli, si estrae dalla pila l'ultimo riferimento memorizzato e si riprende l'esame dall'eventuale nodo fratello di tale nodo;
- se la pila è vuota, la visita dell'albero è terminata.

Avendo preventivamente definito le seguenti classi *Nodo* e *Pila* per elementi di classe *NodoAlbero*

```
public class Nodo {
  private NodoAlbero info;
  private Nodo link;

  public Nodo (NodoAlbero info) {
    this.info = info;
    link = null;
  }

  public void setInfo(NodoAlbero info) {
    this.info = info;
  }
```

```
  public NodoAlbero getInfo() {
    return info;
  }

  public void setLink(Nodo link) {
    this.link = link;
  }

  public Nodo getLink() {
    return link;
  }
}

public class Pila {
  private Nodo head;

  public Pila() {
    head = null;
  }

  public Nodo creaNodo(NodoAlbero nodo, Nodo link) {
    Nodo p = new Nodo(nodo);

    p.setLink(link);
    return p;
  }

  public void push(NodoAlbero nodo) {
    Nodo p;
    p = creaNodo(nodo, head);
    head = p;
  }

  public NodoAlbero pop() {
    Nodo p;

    if (head==null) // pila vuota?
      return null;
    p = head;
    head = head.getLink();
    return p.getInfo;
  }
}
```

il seguente metodo Java implementa l'algoritmo iterativo descritto:

```
public void visitaAnticipata() {
  Pila pila = new Pila();
  NodoAlbero p;
  if (ptr == null) // albero vuoto?
    return;
  p=ptr;
  while (true) {
    esamina(p.getInfo());
    if (p.getFirstChild() != null) {
      pila.push(p); // inserimento riferimento nodo
                    // corrente nella pila
```

```
      p = p.getFirstChild();
    }
    else {
      while (true) {
        if (p.getFirstSibling() != null) {
          p = p.getFirstSibling();
          break;
        }
        else {
          p = pila.pop();  // estrazione nodo dalla pila
          if (p == null)   // pila vuota?
            return;        // fine visita
        }
      }
    }
  }
}
```

Algoritmo di visita in ordine differito

0) Se $n > 0$ è il numero di sottoalberi della radice:
1) Visita il primo sottoalbero in ordine differito;
2) Visita il secondo sottoalbero in ordine differito;
…;
n) Visita l'n-esimo sottoalbero in ordine differito;
$n + 1$) Visita la radice.

Nel linguaggio di programmazione Java l'algoritmo può essere implementato mediante il seguente metodo

```
private void visitaDifferita(NodoAlbero p) {
  if (p == null)
    return;
  if (p.getFirstChild() != null)
    visitaDifferita(p.getFirstChild());
  esamina(p.getInfo());
  if (p.getFirstSibling() != null)
    visitaDifferita(p.getFirstSibling());
}
```

invocato da un metodo pubblico in *overloading* a partire dal nodo radice dell'albero:

```
public void visitaDifferita() {
  visitaDifferita(ptr);
}
```

OSSERVAZIONE Dall'esame dell'algoritmo proposto si può vedere chiaramente che, a differenza di quanto avviene per la visita anticipata, in questo caso, la radice (sia quella principale sia quella di ogni sottoalbero) viene visitata sempre per ultima.

ESEMPIO

Visitando l'albero di FIGURA **15** in modo differito si ottiene la sequenza *e, f, g, b, h, i, c, l, d, a*.

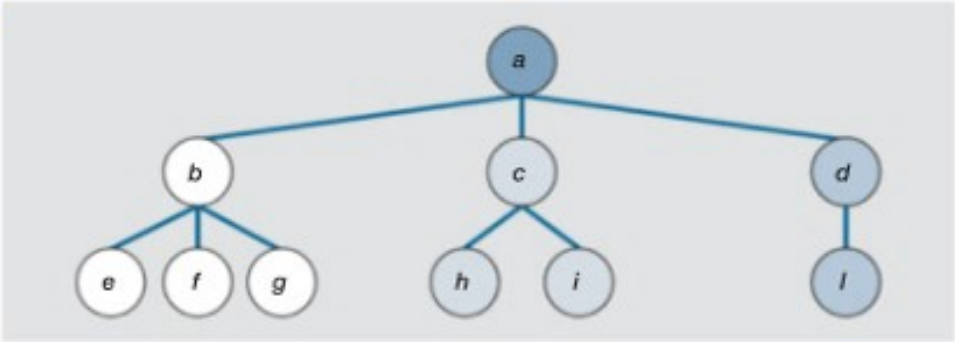


FIGURA **15**

OSSERVAZIONE Gli algoritmi di visita dei nodi di un albero sono facilmente trasformabili in algoritmi di ricerca della presenza o meno nei nodi di un albero di una specifica informazione: è infatti sufficiente confrontare l'informazione con la componente informativa dei nodi visitati.

ESEMPIO

Il codice riportato in questo esempio fornisce una semplice implementazione di alcuni metodi di riferimento per un albero con radice.
Nel *main* vengono fornite le istruzioni necessarie per la memorizzazione dell'albero di FIGURA **16**, che successivamente viene visitato in ordine anticipato per verificarne la corretta costruzione.

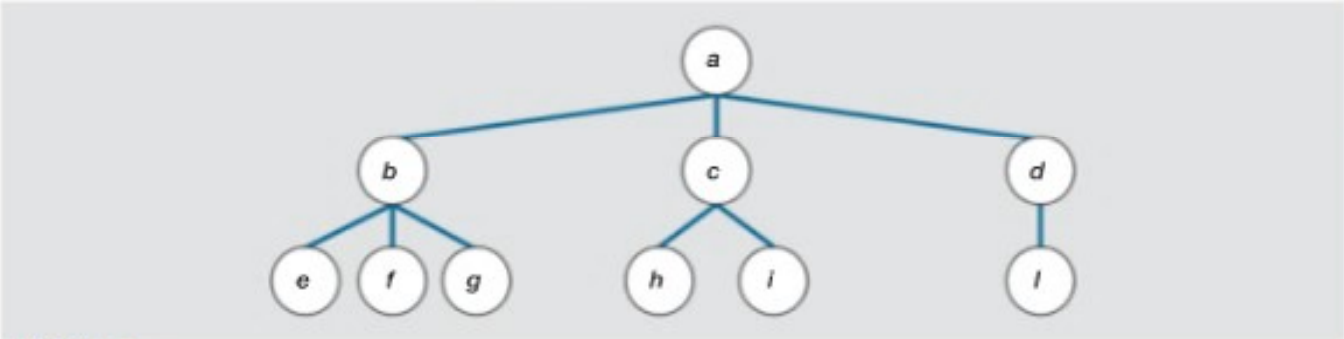


FIGURA **16**

Per semplicità si è supposto che nel nostro albero non vi siano informazioni duplicate e l'attributo *info* della classe *NodoAlbero* vista in precedenza sia definito come oggetto della classe *Character* (classe *wrapper* del tipo primitivo **char**):

```
...
private Character info;
...
```

La classe *Albero* viene definita come segue

```
public class Albero {
private NodoAlbero ptr;

  public Albero() {
    ptr=null;
  }

  public void visitaAnticipata() {
    visita Anticipata(ptr);
  }
```

```
private void visitaAnticipata(NodoAlbero p) {
  if (p == null) return;
  System.out.println(p.getInfo());
  if (p.getFirstChild() != null)
    visitaAnticipata(p.getFirstChild());
  if (p.getFirstSibling() != null)
    visitaAnticipata(p.getFirstSibling());
}

private NodoAlbero cercaNodo(NodoAlbero p,char chiave) {
  NodoAlbero p1 = null;
  if (p == null)
    return null;
  if (p.getInfo()==chiave) return p;   // ricerca con successo
  if (p.getFirstChild() != null) {     // ricerca verso il figlio
    p1 = cercaNodo(p.getFirstChild(),chiave);
    if (p1 != null) return p1; // interruzione della ricorsione
  }
  if (p.getFirstSibling() != null) { // ricerca verso il fratello
    p1=cercaNodo(p.getFirstSibling(),chiave);
    if (p1 != null) return p1; // interruzione della ricorsione
  }
  return null;
}

public void aggiungiFiglio(char chiave, char info) {
  NodoAlbero n = new NodoAlbero(info) ;
  if (ptr == null){ //albero vuoto: inserimento radice
      ptr = n;
      return;
  }
  NodoAlbero p;
  p = cercaNodo(ptr,chiave);
  if (p != null)
      if (p.getFirstChild() == null)
          p.setFirstChild(n); //inserimento primo figlio
      else{
          p = p.getFirstChild();
          while(p.getFirstSibling() != null)
            p = p.getFirstSibling();
          p.setFirstSibling(n); //aggiunta fratello
      }
}

public static void main(String[] args) {
      Albero a = new Albero();

      a.aggiungiFiglio('a','a');

      a.aggiungiFiglio('a','b');
      a.aggiungiFiglio('a','c');
      a.aggiungiFiglio('a','d');

      a.aggiungiFiglio('b','e');
      a.aggiungiFiglio('b','f');
      a.aggiungiFiglio('b','g');

      a.aggiungiFiglio('c','h');
```

```
        a.aggiungiFiglio('c','i');
        a.aggiungiFiglio('d','l');

        a.visitaAnticipata();
    }
}
```

dove:

- l'operazione *esamina* del metodo ricorsivo *ricercaAnticipata* è stata implementata come output a video dell'informazione del nodo visitato;
- il metodo privato *cercaNodo* restituisce il riferimento al nodo di cui viene fornito un carattere come chiave di ricerca; se il nodo non esiste il metodo ritorna **null**. Si noti come questo sia stato sviluppato effettuando la ricerca sulla base dell'ordine di visita anticipato. Il metodo *cercaNodo* è stato implementato modificando il codice usato per il metodo *visitaAnticipata*, interrompendo la ricorsione non appena la ricerca ha successo;
- il metodo *aggiungiFiglio* prevede due parametri: il primo relativo al nodo su cui si intende operare e il secondo la nuova informazione da inserire (per esempio il significato dell'istruzione *a.aggiungiFiglio('b', 'e');* è: nell'ambito dell'albero *a* inserisci l'informazione *'e'* in un nuovo nodo figlio del nodo la cui informazione è *'b'*).

Dato lo scopo dimostrativo del codice, non è stato implementato alcun controllo per prevenire l'inserimento di informazioni duplicate che, date le premesse e la modalità di memorizzazione dell'albero, invaliderebbe il funzionamento del medesimo.

L'output prodotto dal *main* è

```
a
b
e
f
g
c
h
i
d
l
```

che rappresenta appunto la sequenza di nodi che si ottiene visitando in ordine anticipato l'albero utilizzato nell'esempio.

5.2 Albero binario

Un **albero binario** (BT, *Binary Tree*) è un albero in cui ogni nodo ha al massimo due figli, denominati rispettivamente **figlio sinistro** e **figlio destro**.

Un albero binario come quello di FIGURA **17** può essere memorizzato con una struttura del tipo mostrato in FIGURA **18**, dove:

- *ptr* è il riferimento al nodo radice;
- ogni nodo è composto da tre elementi: la componente informativa (*info*), il riferimento al figlio sinistro (*psx*) e il riferimento al figlio destro (*pdx*).

OSSERVAZIONE Un albero binario può essere costituito anche da un'insieme vuoto di nodi, per cui dato un albero binario *ab*, se si elimina la radice, si ottengono due insiemi *ab1* e *ab2* ognuno dei quali è ancora un albero binario.

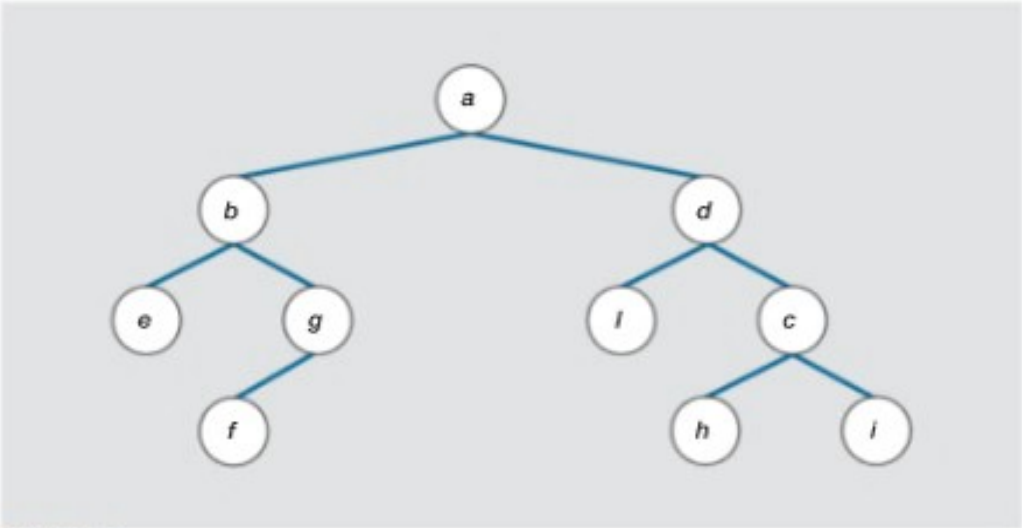


FIGURA **17**

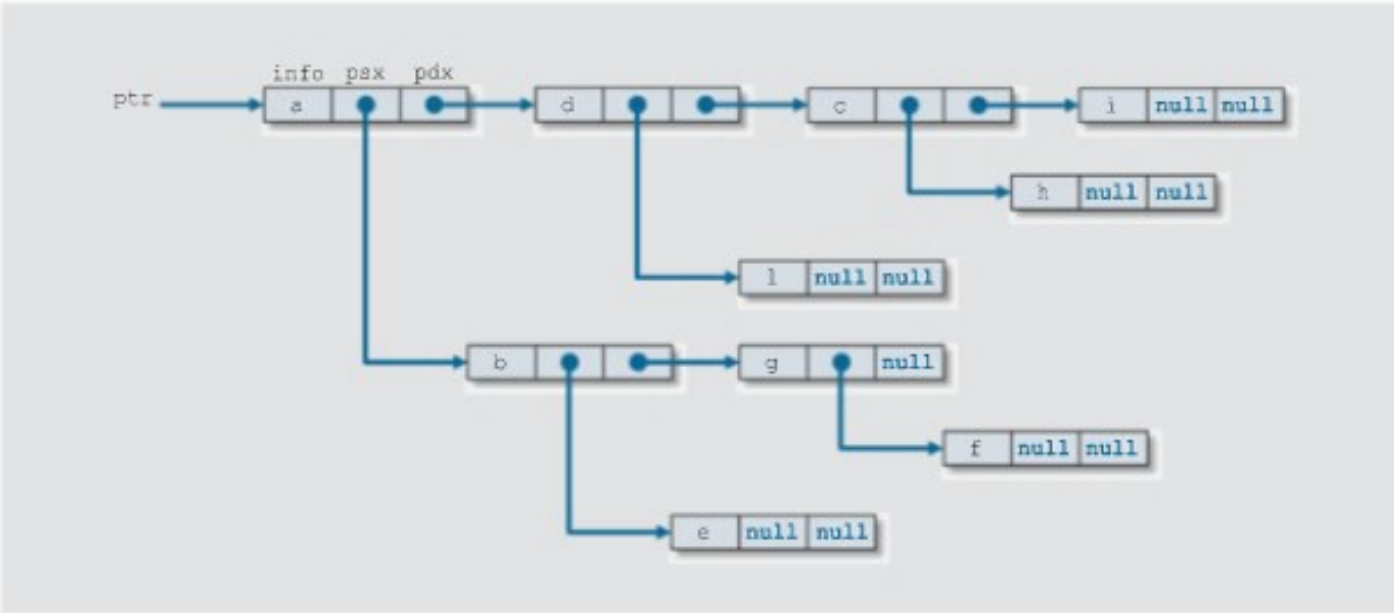


FIGURA **18**

ESEMPIO

Tralasciando la definizione delle operazioni, il diagramma UML delle classi che consentono di rappresentare un albero binario è come quello di FIGURA **19**.

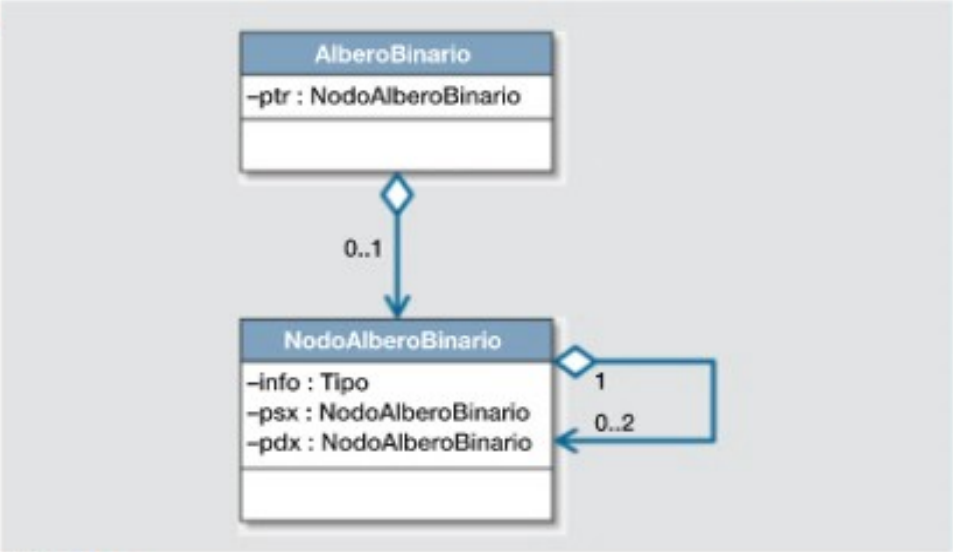


FIGURA **19**

Nel linguaggio di programmazione Java la classe *NodoAlberoBinario* è di immediata realizzazione (è stato lasciato non definito il tipo della componente informativa):

```
public class NodoAlberoBinario {

  private ... info;
  private NodoAlberoBinario psx;
  private NodoAlberoBinario pdx;

  public NodoAlberoBinario(... info) {
    this.info = new ...(info);
    psx = null;
    pdx = null;
  }

  public void setInfo(... info) {
    this.info = new ...(info);
  }

  public ... getInfo() {
    return new ...(info);
  }

  public void setLeftChild(NodoAlberoBinario psx) {
    this.psx = psx;
  }

  public NodoAlberoBinario getLeftChild() {
    return psx;
  }

  public void setRightChild(NodoAlberoBinario pdx) {
    this.pdx = pdx;
  }

  public NodoAlberoBinario getRightChild() {
    return pdx;
  }
}
```

La classe che rappresenta l'albero binario avrà come attributo il riferimento al nodo radice:

```
public class AlberoBinario {
  private NodoAlberoBinario ptr;
  ...
  ...
  ...
}
```

In un albero binario, oltre alle visite in ordine anticipato e differito, è possibile condurre una visita in ordine simmetrico.

ESEMPIO I seguenti metodi Java consentono di visitare un albero binario rispettivamente in ordine anticipato e differito:

```
private void visitaAnticipata(NodoAlberoBinario p) {
  if (p == null)
    return;
  esamina(p.getInfo());
```

```
    if (p.getLeftChild() != null)
      visitaAnticipata(p.getLeftChild());
    if (p.getRightChild() != null)
      visitaAnticipata(p.getRightChild());
  }

  public void visitaAnticipata() {
    visitaAnticipata(ptr);
  }

  private void visitaDifferita(NodoAlberoBinario p) {
    if (p == null)
      return;
    if (p.getLeftChild() != null)
      visitaDifferita(p.getLeftChild());
    if (p.getRightChild() != null)
      visitaDifferita(p.getRightChild());
    esamina(p.getInfo());
  }

  public void visitaDifferita() {
    visitaDifferita(ptr);
  }
```

Visitando in ordine anticipato l'albero binario di FIGURA **20**, si ottiene la sequenza di nodi *a, b, e, g, f, d, l, c, h, i*; la visita in ordine differito produce invece la sequenza di nodi *e, f, g, b, l, h, i, c, d, a*.

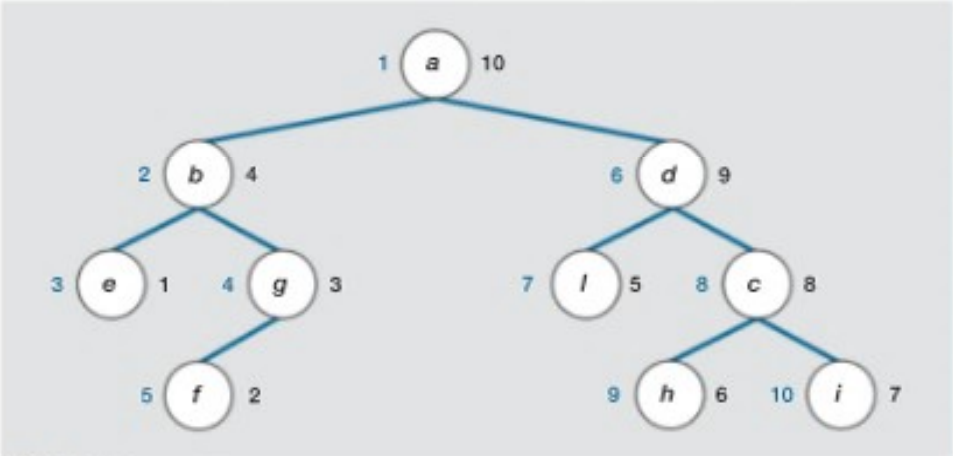


FIGURA **20**

Algoritmo di visita in ordine simmetrico

1) Visita il sottoalbero del figlio sinistro in ordine simmetrico.
2) Visita la radice.
3) Visita il sottoalbero del figlio destro in ordine simmetrico.

Nel linguaggio di programmazione Java l'algoritmo può essere implementato mediante il seguente metodo

```
private void visitaSimmetrica(NodoAlberoBinario p) {
  if (p == null)
    return;
  if (p.getLeftChild() != null)
    visitaSimmetrica(p.getLeftChild());
```

```
  esamina(p.getInfo());
  if (p.getRightChild() != null)
  visitaSimmetrica(p.getRightChild());
}
```

invocato da un metodo pubblico in *overloading* a partire dal nodo radice dell'albero:

```
public void visitaSimmetrica() {
  visitaSimmetrica(ptr);
}
```

ESEMPIO La visita in ordine simmetrico dell'albero binario dell'esempio precedente genera la sequenza di nodi *e, b, f, g, a, l, d, h, c, i*.

È possibile trasformare un albero generico A in un albero binario B applicando l'algoritmo seguente:

1) la radice di B coincide con quella di A;
2) per ogni altro nodo n di B, la radice del sottoalbero sinistro coincide con il primo figlio che il nodo n ha in A, mentre la radice del sottoalbero destro coincide con il primo fratello che il nodo n ha in A.

ESEMPIO Con la precedente trasformazione l'albero generico di FIGURA **21** diviene l'albero binario di FIGURA **22**.

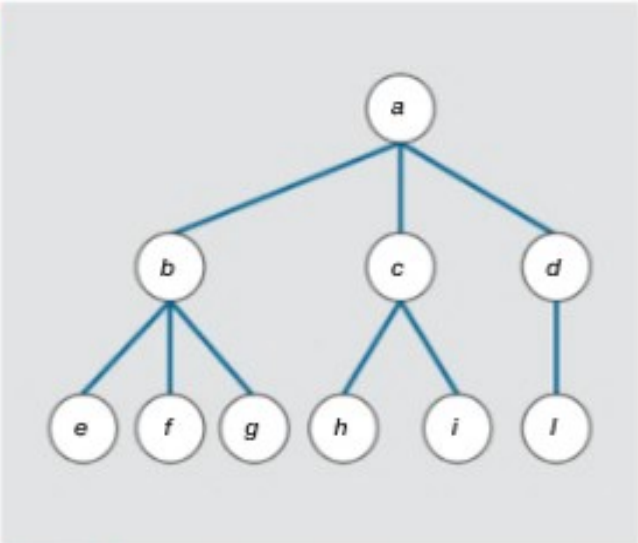


FIGURA **21**

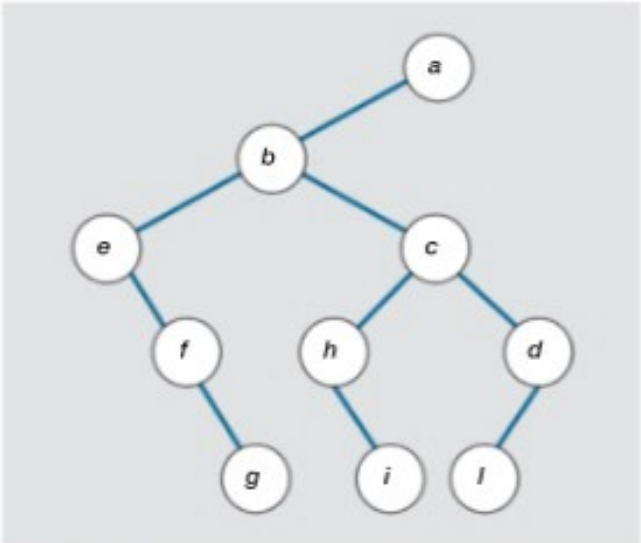


FIGURA **22**

OSSERVAZIONE A seguito delle regole che lo definiscono, l'algoritmo di trasformazione produce sempre un albero binario in cui la radice è priva del sottoalbero destro. Visitando in ordine simmetrico l'albero binario risultante si ottiene la stessa sequenza di nodi che si otterrebbe visitando in ordine differito l'albero generico originale. La trasformazione di un albero generico in un albero binario viene spesso operata per rendere più semplici le operazioni di gestione dell'insieme di dati rappresentato.

5.3 Alberi binari di ricerca

Un albero binario è definito **albero binario di ricerca** (BST, *Binary Search Tree*) se per ogni nodo *n* dell'albero i nodi del sottoalbero sinistro contengono – secondo un criterio di ordinamento prefissato – solo valori minori del valore contenuto nel nodo *n* e i nodi del sottoalbero destro contengono solo valori maggiori di questo.

ESEMPIO

Volendo realizzare un albero binario di ricerca dove la componente informativa dei nodi è per semplicità un singolo carattere, la seguente classe rappresenta un nodo dell'albero:

```
public class NodoAlberoBinario {

  private char info;
  private NodoAlberoBinario psx;
  private NodoAlberoBinario pdx;

  public NodoAlberoBinario(char info) {
    this.info = info;
    psx = null;
    pdx = null;
  }

  public void setInfo(char info) {
    this.info = info;
  }
  public char getInfo() {
    return info;
  }

  public void setLeftChild(NodoAlberoBinario psx) {
    this.psx = psx;
  }

  public NodoAlberoBinario getLeftChild() {
    return psx;
  }

  public void setRightChild(NodoAlberoBinario pdx) {
    this.pdx = pdx;
  }

  public NodoAlberoBinario getRightChild() {
    return pdx;
  }
}
```

Se nella classe che rappresenta l'albero binario di ricerca il riferimento al nodo radice è

```
private NodoAlberoBinario ptr;
```

i seguenti metodi consentono di aggiungere un nodo all'albero in modo ricorsivo:

```
private void aggiungiNodo(NodoAlberoBinario n, NodoAlberoBinario p) {
  if (n.getInfo()<p.getInfo()) {
    if (p.getLeftChild() == null)
      p.setLeftChild(n);
    else
      aggiungiNodo(n, p.getLeftChild());
  }
```

```
    else { // n.getInfo()>=p.getInfo()
      if (p.getRightChild() == null)
        p.setRightChild(n);
      else
        aggiungiNodo(n, p.getRightChild());
    }
  }

  public void aggiungiNodo(char info) {
    NodoAlberoBinario n = new NodoAlberoBinario(info);
    if (ptr == null)
      ptr = new NodoAlberoBinario(info);
    else
      aggiungiNodo(n, ptr);
  }
```

OSSERVAZIONE Il metodo *aggiungiNodo* dell'esempio precedente può essere così implementato in modo iterativo:

```
public void aggiungiNodo(char info) {
  NodoAlberoBinario n = new NodoAlberoBinario(info);
  NodoAlberoBinario nodoCorrente, nodoPrecedente;

  if (ptr == null)
    ptr = new NodoAlberoBinario(info);
  else {
    nodoPrecedente = ptr;
    if (info<nodoPrecedente.getInfo())
      nodoCorrente = nodoPrecedente.getLeftChild();
    else
      nodoCorrente = nodoPrecedente.getRightChild();
    while (nodoCorrente != null) {
      nodoPrecedente = nodoCorrente;
      if (info<nodoPrecedente.getInfo())
        nodoCorrente = nodoPrecedente.getLeftChild();
      else
        nodoCorrente = nodoPrecedente.getRightChild();
    }
    if (info<nodoPrecedente.getInfo())
      nodoPrecedente.setLeftChild(n);
    else
      nodoPrecedente.setRightChild(n);
  }
}
```

Visitando in ordine simmetrico un albero binario di ricerca si ottiene una sequenza di nodi in accordo con il criterio di ordinamento prefissato (per esempio, supponendo che i nodi contengano dei caratteri alfabetici, in ordine alfabetico).

ESEMPIO

L'albero binario di FIGURA **23** è un albero binario di ricerca secondo il criterio dell'ordinamento alfabetico.

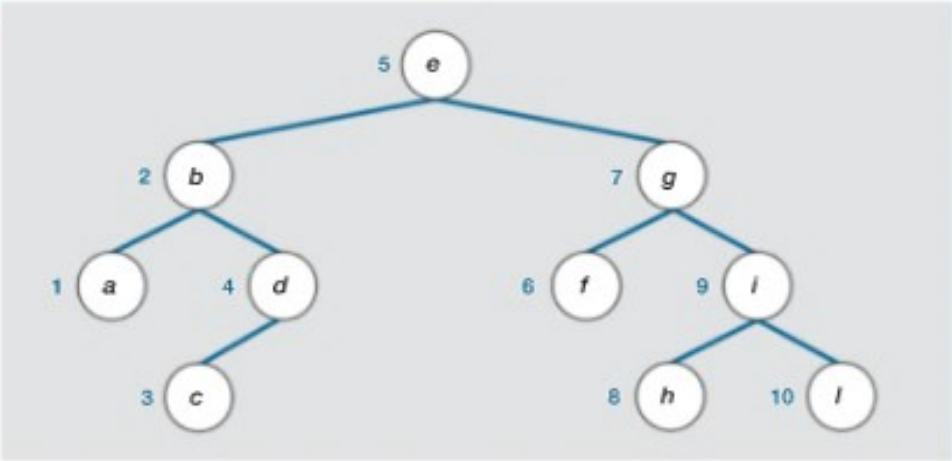


FIGURA **23**

La visita dei nodi in ordine simmetrico produce la sequenza *a, b, c, d, e, f, g, h, i, l*.

Gli alberi binari di ricerca sono prevalentemente utilizzati per la memorizzazione di informazioni in base alle quali è successivamente necessario effettuare ricerche in tempi rapidi. Un'operazione di ricerca in una struttura dati consiste nel verificare se un'informazione è o meno contenuta nelle componenti informative dei nodi[1].

Per la particolare strutturazione dell'albero binario di ricerca l'algoritmo non deve effettuarne una visita completa, ma selezionare a ogni passo il solo sottoalbero che può eventualmente contenere l'informazione ricercata. La seguente è una possibile implementazione in linguaggio Java

```
private boolean ricerca(NodoAlberoBinario p, char info) {
  if (p == null)
    return false;
  if (ricerca(p.getLeftChild(), info);
    return true;
  if (p.getInfo() == info)
    return true;
  if (ricerca(p.getRightChild(), info);
    return true;
  return false;
}

public boolean ricerca(char info) {
  return ricerca(ptr, info);
}
```

dove *ptr* è il riferimento al nodo radice.

1. Spesso il confronto avviene su una sola parte della componente informativa (la «chiave» di ricerca) e il risultato della ricerca è una copia della componente informativa stessa che comprende le informazioni associate alla chiave (per esempio in una rubrica si ricerca il numero di telefono relativo a un nominativo utilizzando il nominativo stesso come chiave di ricerca).

OSSERVAZIONE Il metodo *ricerca* può essere così implementato in modo iterativo:

```
public boolean ricerca(int info) {
   NodoAlberoBinario p = ptr;

   while (p != null) {
```

```
      if ((p.getInfo() == info)
        return true;
      if (p.getInfo()<info)
        p = p.getLeftChild();
      else
        p = p.getRightChild();
    }
    return false;
  }
```

ESEMPIO

L'applicazione dell'algoritmo per la ricerca del carattere *a* all'albero binario di ricerca dell'esempio precedente porta all'analisi dei nodi evidenziati (FIGURA **24**).

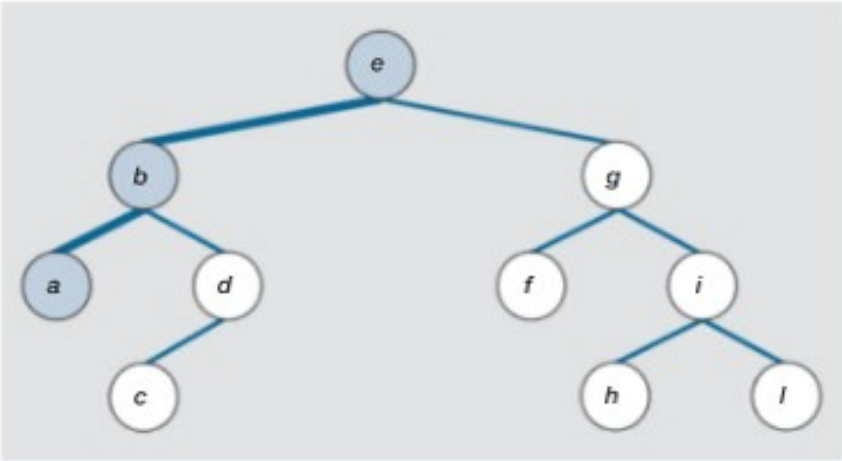


FIGURA **24**

In generale la ricerca di un'informazione in un albero binario di ricerca richiede al più di prendere in esame il numero di nodi presenti sul più lungo percorso radice-foglia dell'albero.

Un albero binario di ricerca è **bilanciato** se l'albero stesso e ogni suo sottoalbero hanno un numero di nodi presenti nel sottoalbero sinistro che differisce al più di 1 dal numero di nodi del sottoalbero destro.

In un albero binario di ricerca bilanciato con n nodi la profondità di una foglia, cioè la sua distanza dalla radice, è al massimo

$$\log_2 n$$

Questo valore rappresenta il massimo numero di confronti necessari per la ricerca di un'informazione nell'albero ed equivale al numero di confronti necessari per la ricerca di un elemento in un vettore ordinato utilizzando l'algoritmo dicotomico.

Il vantaggio dell'uso di un albero binario di ricerca è conseguenza del fatto che le informazioni che contiene mantengono l'ordinamento anche in presenza di operazioni di inserimento e/o di eliminazione di nodi. Le operazioni di inserimento possono produrre sbilanciamenti nell'albero che tendono ad allungare i percorsi radice-foglie deteriorando le prestazioni dell'algoritmo di ricerca; le operazioni di eliminazione possono richiedere lo spostamento di molti nodi per mantenere i vincoli di ordinamento.

Operare con un albero binario di ricerca richiede di conseguenza un codice complesso che ristrutturi l'albero in coincidenza di inserimenti e/o eliminazioni di nodi, allo scopo di mantenere l'ordinamento dei nodi in base al criterio dato e il bilanciamento dell'albero stesso, perché un albero fortemente sbilanciato rende l'algoritmo di ricerca poco efficiente.

ESEMPIO

Costruendo un albero binario di ricerca inserendo i nodi in modo sequenziale rispetto al criterio di ordinamento prefissato si ottiene un albero degenere che coincide di fatto con una lista che rende implicitamente l'algoritmo di ricerca una scansione sequenziale dei nodi (FIGURA **25**).

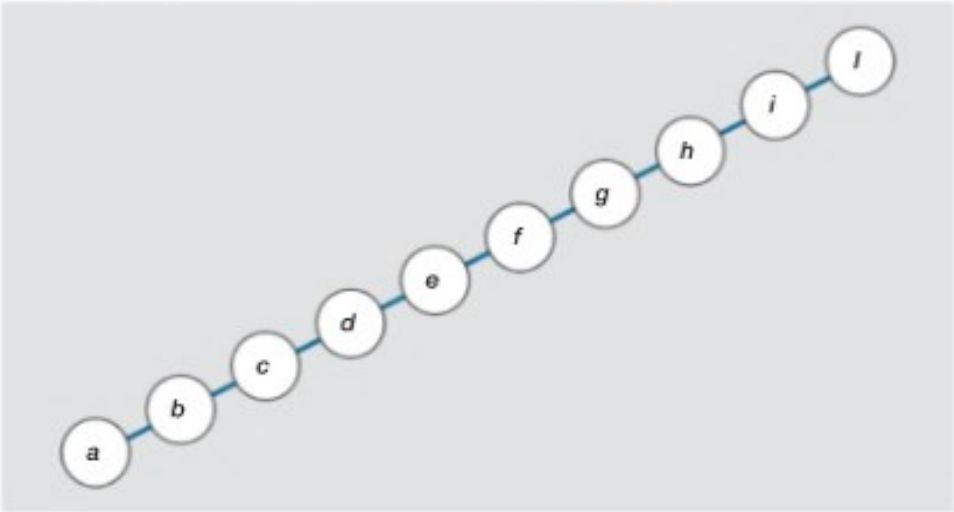


FIGURA **25**

6 Tabelle e indirizzamento *hash*

Gli elenchi di dati sono tra le strutture informative più utilizzate nelle applicazioni informatiche e le problematiche che attengono alla loro gestione sono sempre state oggetto di studi e ricerche finalizzate a migliorane l'efficienza: le **tabelle** costituiscono una modalità di implementazione degli elenchi.

Una **tabella** è una successione di elementi informativi, ciascuno dei quali è composto da due parti logicamente distinte:

- la **chiave** *k*;
- l'**informazione** *i* associata alla chiave.

ESEMPIO

In un dizionario Inglese-Italiano il vocabolo in lingua inglese costituisce la chiave, mentre il corrispondente vocabolo – o i corrispondenti vocaboli – in lingua italiana è l'informazione che viene normalmente ricercata e acceduta a partire dalla chiave.

Nel linguaggio di programmazione Java un modo per realizzare una tabella di elementi del tipo specificato da una determinata classe *Tipo* consiste nel definire una nuova classe che ha come attributo un *array* di elementi di classe *Tipo*: il ricorso a un *array* garantisce l'accesso diretto ai singoli elementi.

Hash

Nella cultura americana il termine *hash* viene utilizzato per indicare una polpetta di carne e verdura realizzata triturando e mescolando gli avanzi del giorno precedente.
Per analogia il termine *hash address* è l'indirizzo che si ottiene «triturando» e «mescolando» opportunamente il valore di una chiave.

L'informazione associata alla chiave nella classe *Tipo* può essere costituita da più campi; per esempio, nel caso di una persona, il cognome, il nome, il sesso, ...

L'operazione fondamentale che si applica a una tabella è la **ricerca** di un elemento – che comprende la componente di informazione – a partire dal valore della chiave.

Per ogni elemento della tabella si definisce **lunghezza di ricerca** il numero di chiavi esaminate nel corso della ricerca della chiave associata all'elemento stesso. Un parametro caratteristico di un metodo di ricerca è la **lunghezza media di ricerca**, definita come la media delle lunghezze di ricerca di tutti gli elementi di una tabella. Allo scopo di velocizzare le operazioni di ricerca è ovviamente importante che tale valore sia il più basso possibile.

La ricerca talvolta è funzionale ad altre operazioni; per esempio, è chiaro che l'eliminazione di un elemento presuppone la sua preventiva ricerca, mentre in alcune situazioni l'inserimento di un nuovo elemento può essere condizionato dal fatto che la sua chiave non sia già presente nella tabella.

Se gli n elementi di una tabella non sono ordinati in funzione della chiave, l'unica possibilità di ricerca è la ricerca sequenziale completa, che ha una lunghezza media di ricerca proporzionale a $n/2$. Se invece gli elementi della tabella sono ordinati, è possibile applicare algoritmi di ricerca molto più performanti, come la ricerca binaria, che ha una lunghezza media di ricerca proporzionale a $\log_2 n$.

Una tabella può sempre essere ordinata in funzione del campo chiave applicando a essa un qualsiasi algoritmo di ordinamento, ma – dato che un qualsiasi algoritmo di ordinamento è molto «costoso» in termini di tempo di elaborazione – questa soluzione è praticabile solo se la variazione della tabella stessa (eliminazioni di elementi esistenti e/o inserimenti di nuovi elementi) non è frequente.

I metodi di ricerca tradizionali (sequenziale e binario) non sfruttano la possibilità di ricavare direttamente dalla chiave la massima informazione circa la posizione del relativo elemento nella tabella.

Determinare la posizione di un elemento nell'*array* che implementa la tabella in funzione della sua chiave rende la lunghezza media di ricerca praticamente costante e indipendente dal numero n di elementi della tabella. Questo vantaggio è tanto più evidente quanto più grande è n.

Per conseguire l'obiettivo di «calcolare» la posizione di un elemento nella tabella a partire dalla sua chiave, è necessario definire una funzione

d'accesso h che, applicata al valore della chiave k, fornisca un valore intero p che rappresenta la posizione dell'elemento nel vettore:

$$p = h(k)$$

Le funzioni con le caratteristiche adatte per questo impiego sono note come **funzioni *hash***[2].

2. Una specifica categoria di funzioni *hash* è utilizzata nel campo della crittografia: in questo caso, al fine di garantire la sicurezza degli algoritmi in cui sono utilizzate, sono richiesti requisiti estremamente stringenti; una funzione *hash* di accesso non dovrebbe mai essere utilizzata come funzione *hash* crittografica e viceversa.

ESEMPIO

Volendo gestire in forma di tabella l'elenco degli invitati a una festa individuando il nome come chiave, è possibile utilizzare come funzione *hash* il seguente metodo Java

```
private int hash(String chiave) {
  int p=0, n=chiave.length();
  for (int i=0; i<n; i++)
    p += chiave.charAt(i);
  return p;
}
```

che somma i valori numerici dei singoli caratteri della stringa che contiene il nome dell'invitato. In questo caso la funzione h è così definita

$$h(k) = \sum_{i=0}^{n-1} chiave[i]$$

dove *chiave*[*i*] indica il valore dell'*i*-esimo carattere della chiave e il valore n è il numero di caratteri della chiave stessa. Il valore numerico di un carattere coincide con la codifica numerica Unicode del carattere stesso. Se il nominativo è «Rossi Mario» si ha

$$h(\text{"Rossi Mario"}) =$$

$$= 82 + 111 + 115 + 115 + 105 + 32 + 77 + 97 + 114 + 105 + 111 = 1064$$

Dato che il valore numerico generato dalla funzione dell'esempio precedente potrebbe essere molto grande e che la tabella è una struttura dati con un numero prefissato di elementi, è opportuno integrare la funzione *hash* determinando il modulo del risultato rispetto alla dimensione d della tabella stessa:

$$h(k) = \left(\sum_{i=0}^{n-1} chiave[i] \right) \bmod d$$

In Java il metodo diviene

```
private int hash(String chiave) {
  int p=0;
  for (int i=0; i<chiave.length(); i++)
    p += chiave.charAt(i);
  p = p % tabella.length;
  return p;
}
```

dove *tabella* è l'attributo della classe che implementa l'*array* di elementi. Se *tabella* è stata dimensionata per 100 elementi in corrispondenza del nominativo «Rossi Mario» si avrà

Metodo *hashCode* in Java

Tutte le classi definite in un programma in linguaggio Java ereditano il metodo *hashCode*, che fornisce una rudimentale funzione *hash* per gli oggetti istanziati. In particolare, il metodo *hashCode* della classe *String* determina un valore di tipo `int` con un algoritmo simile a quello riportato nell'esempio.

Una buona pratica di programmazione Java prevede la ridefinizione di un metodo *hashCode* per ogni classe definita che implementi una funzione *hash* adattata al numero e alla tipologia degli attributi della classe stessa: l'uguaglianza di due oggetti è determinata dall'operatore «==» in base all'uguaglianza dei valori restituiti dal metodo *hashCode* della classe di cui gli oggetti stessi sono istanza.

$$h(\text{"Rossi Mario"}) =$$

$$= (82 + 111 + 115 + 115 + 105 + 32 + 77 + 97 + 114 + 105 + 111) \bmod 100 = 64$$

e le informazioni relative saranno memorizzate nella posizione di indice 64 dell'*array*.

Non esiste un criterio standard per definire una funzione *hash* che, in pratica, viene progettata di volta in volta a seconda del tipo di problema da risolvere. La definizione di una valida funzione *hash* è infatti influenzata da parametri quali la tipologia e il numero delle chiavi.

La validità di una funzione *hash* è data da un insieme di fattori tra cui:

- il **basso costo** di computazione in termini di tempo e di risorse per il calcolo;
- il **determinismo** richiesto dal fatto che la funzione deve sempre calcolare la stessa posizione p per la stessa chiave k;
- l'**uniformità** della distribuzione dei valori delle posizioni che non devono essere concentrati su alcuni valori rispetto ad altri.

La progettazione di algoritmi per l'implementazione di funzioni *hash* valide è di conseguenza un compito estremamente specialistico che non deve essere improvvisato: il programmatore che necessita di utilizzarle troverà con facilità mediante una ricerca in rete, o l'acquisizione della letteratura tecnica specifica, la funzione più adatta ai propri scopi.

Quando si trattano le funzioni *hash* si fa in genere riferimento a una **tecnica di indirizzamento** che è necessario distinguere dal concetto di **metodo di ricerca** (come, per esempio, la ricerca binaria). I metodi di ricerca hanno il solo scopo di ricercare le chiavi all'interno di una tabella di elementi, mentre le tecniche di indirizzamento sono necessariamente usate sia per l'operazione di memorizzazione degli elementi nella tabella (infatti è indispensabile determinare la posizione della tabella in cui memorizzare l'elemento) sia per l'operazione di ricerca di un elemento a partire dalla chiave.

Poiché in una tabella *hash* non vi è alcun modo di accedere alle informazioni in essa contenute se non attraverso la sua funzione di *hashing*, non è possibile definire su queste un qualsiasi criterio di ordinamento (per esempio, nomi in ordine alfabetico) interno alla tabella.

6.1 La gestione delle collisioni

Se la tabella ha d elementi è auspicabile che le seguenti condizioni siano sempre verificate:

- $h(k) = p$ con p intero e tale che $0 \leq p < d$;
- $h(k_1) \neq h(k_2)$ per ogni coppia di chiavi k_1, k_2 distinte.

Se le condizioni sopra riportate fossero entrambe valide per qualsiasi chiave k, la lunghezza media di ricerca per la funzione h sarebbe uguale a 1, il che ovviamente è il valore minimo possibile.

La prima condizione è, come abbiamo visto, di facile soddisfazione applicando al calcolo vero e proprio della funzione *hash* un'operazione di modulo rispetto alla dimensione d della tabella. Invece la seconda condizione – almeno nei casi di interesse pratico dove il numero di chiavi potenziali distinte è estremamente elevato e in ogni caso molto maggiore della dimensione d della tabella – è di difficile soddisfazione, volendo mantenere una relativa semplicità di calcolo.

In pratica il fenomeno delle «collisioni» è inevitabile.

Si definisce **collisione** il risultato dell'applicazione a chiavi distinte di una funzione *hash* che genera lo stesso risultato (il che determina la stessa posizione di memorizzazione nella tabella per due elementi diversi).

Una gestione efficiente delle collisioni è fondamentale per l'impiego pratico di una tabella con indirizzamento *hash*. Il primo accorgimento da realizzare è la minimizzazione della probabilità che una collisione avvenga: a questo scopo occorre avere una funzione *hash* che trasformi le chiavi in indirizzi nel modo più uniforme possibile sfruttando tutte le posizioni utili della tabella e, spesso, è anche necessario dimensionare la tabella di un numero di posizioni maggiore rispetto al numero di elementi da memorizzare.

Si definisce **fattore di carico** (*lf*, *load factor*) di una tabella con indirizzamento *hash* il rapporto tra il numero n di elementi effettivamente memorizzati e la dimensione d della tabella (cioè il massimo numero di elementi memorizzabili):

$$lf = \frac{n}{d}$$

L'efficienza dell'indirizzamento *hash* è direttamente legata al numero di collisioni che si generano: un basso fattore di carico è, dopo la scelta della funzione h, il primo requisito da rispettare.
Ovviamente bassi fattori di carico hanno come conseguenza uno spreco della memoria allocata per la tabella.

Anche con fattori di carico bassi la probabilità che avvenga una collisione non è nulla, per cui l'eventualità, pena una certamente indesiderata perdita di dati, deve essere comunque gestita.

Esistono due tipologie di metodi per risolvere il problema delle collisioni: il *concatenamento* e l'*indirizzamento aperto*.

- **Concatenamento**: la tabella è di fatto realizzata come un *array* di liste e ogni posizione della tabella è una lista in cui sono memorizzati tutti gli elementi che hanno la chiave che collide sulla posizione stessa. In

Hashing doppio

Una possibile implementazione del metodo dell'indirizzamento aperto è il cosidetto ***hashing* doppio**, in cui la sequenza delle posizioni da scansionare nel caso che la funzione *hash* primaria h_1 causi una collisione per la chiave k è data dalla seguente formula, che impiega anche una diversa funzione *hash* secondaria h_2

$$p_i = [h_1(k) + i \cdot h_2(k)] \bmod d$$

dove $i = 0, 1, 2, \ldots$ è l'indice dei successivi tentativi di trovare una posizione libera.

altre parole, nella posizione p della tabella è contenuta la lista di tutti gli elementi per cui $h(k)$, dove k è la chiave dell'elemento, è uguale a p. Elementi distinti con chiavi diverse k_1, k_2, ... per cui $h(k_1)$, $h(k_2)$, ... generano in collisione lo stesso valore p sono inseriti nella stessa lista.

ESEMPIO Supponiamo di voler inserire in una tabella di dimensione 5 alcuni elementi le cui chiavi sono i numeri interi 42, 15, 100, 25, 73, 68, 84, 57, 3, 99, 27, 62. Se la funzione *hash* è realizzata banalmente effettuando un'operazione di modulo 5 direttamente sul valore della chiave, la situazione che si ottiene è quella mostrata in FIGURA **26**.

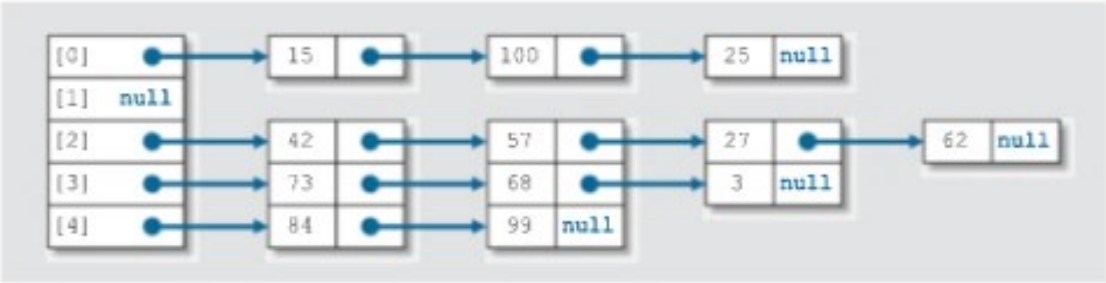


FIGURA **26**

Nell'esempio precedente la funzione *hash* permette l'accesso diretto alla posizione dell'*array* relativa alla chiave a cui è stata applicata, mentre l'individuazione della chiave vera e propria (se esiste) e del relativo elemento prevede la scansione sequenziale della lista associata.

- **Indirizzamento aperto**: in questo caso tutti gli elementi sono contenuti nell'*array* che implementa la tabella; se l'inserimento di un nuovo elemento con chiave k deve avvenire in una posizione $p = h(k)$ che risulta occupata, esso viene memorizzato in una diversa posizione libera secondo una legge predefinita, in cui l'indice di posizione viene sottoposto a un'operazione di modulo rispetto alla dimensione della tabella stessa (in questo modo è come se l'*array* fosse «circolare»: la prima posizione viene considerata consecutiva all'ultima).

ESEMPIO Una delle forme più utilizzate di indirizzamento aperto è la **scansione lineare** che, nella sua realizzazione più semplice, prevede di ricercare una posizione libera tra quelle individuate dalla seguente sequenza

$$p_i = [h(k) + q \cdot i] \bmod d$$

dove $i = 0, 1, 2, \ldots$ è l'indice dei successivi tentativi di trovare una posizione libera e q è un valore costante intero; inoltre i valori q e d devono essere primi tra loro. Se $q = 1$ la scansione ha passo unitario, se $q > 1$ la distanza tra le successive posizioni scansionate è il valore q; in ogni caso la sequenza è a passo fisso e interessa tutte le posizioni dell'*array* prima di ritornare su posizioni già visitate (ciò viene garantito dal fatto che q e d sono primi tra loro).
Per una tabella per cui $d = 1000$ e $q = 3$, se l'applicazione $h(k)$ della funzione *hash* h alla chiave k di un nuovo elemento da inserire genera la posizione $p = 101$ che risulta occupata, sono scansionate le posizioni $101 + 3 = 104$, $101 + 6 = 107$, $101 + 9 = 110$, ... fino a che una di esse non risulta libera.

Utilizzando la scansione lineare è bene dimensionare l'*array* in modo che il fattore di carico non superi 0,75. Infatti è dimostrato che ogni incremento oltre tale valore contribuisce a peggiorare significativamente l'efficienza della gestione. Con la scansione lineare tendono infatti a formarsi *cluster* (agglomerati) di elementi in conseguenza del fatto che più chiavi hanno lo stesso indirizzo *hash* e che posizioni generate dalla funzione *hash* risultano occupate da elementi la cui chiave genera una diversa posizione risultata in precedenza occupata. Gli agglomerati tendono a crescere con il fattore di carico della tabella e possono degradare notevolmente le prestazioni della funzione *hash* utilizzata, rallentando le operazioni di ricerca e inserimento di elementi nella tabella.

ESEMPIO

Una forma alternativa di indirizzamento aperto è la scansione quadratica, che permette di realizzare scansioni a passo variabile. Nella sua forma più semplice prevede di ricercare una posizione libera tra quelle individuate dalla seguente sequenza

$$p_i = [h(k) + q \cdot i^2] \bmod d$$

dove $i = 0, 1, 2, \ldots$ è l'indice dei successivi tentativi di trovare una posizione e q è un valore costante intero; il valore d deve essere in questo caso un numero primo. Il calcolo delle posizioni successive in caso di collisione risulta piuttosto semplice, ma la sequenza generata coinvolge solo circa la metà delle posizioni dell'*array* ($(d - 1)/2$ posizioni oltre a quella iniziale): questo apparente svantaggio è comunque trascurabile su tabelle non troppo piccole.

ESEMPIO

Il diagramma UML di FIGURA **27** rappresenta la classe *TabellaInvitati* per gestire gli inviti a una festa tramite una tabella a indirizzamento *hash*.

FIGURA **27**

La classe è implementata dal seguente codice Java:

```
public class TabellaException extends Exception {
  private String error="";

  TabellaException(String error) {
    this.error = error;
  }

  String getError() {
```

```
      return error;
    }
  }

  public class TabellaInvitati {
    private Invitato festa[]; // tabella
    private int numeroInvitati;

    public TabellaInvitati(int dimensione) {
      festa = new Invitato[dimensione];
      numeroInvitati=0;
    }

    // funzione hash
    private int hash (String chiave) {
      int p = 0;

      for (int i=0; i<chiave.length(); i++) {
        p += chiave.charAt(i);
      }
      return (p % festa.length);
    }

    // operazione R = ricerca per informazioni
    // operazione I = ricerca per inserimento
    private int cercaInvitato(String chiave, char operazione)
                throws TabellaException {
      int p = hash(chiave);

      if (operazione == 'R') {
        // ricerca per informazioni
        if (festa[p] == null)
          throw new TabellaException("Elemento "+chiave+ " non esistente");
        if (festa[p].getNome().equals(chiave))
          return p;
      }
      else {
        // ricerca per inserimento
        if (festa[p] == null)
          return p;
        if (festa[p].getNome().equals(chiave))
          throw new TabellaException("Elemento "+festa[p].getNome()+ " preesistente");
      }
      // collisione -> scansione lineare
      p = scansioneLineare(p, chiave, operazione);
      return p;
    }

    public Invitato cerca(String nome) throws TabellaException {
      int p = cercaInvitato(nome,'R');
      return new Invitato(festa[p]);
    }

    public int getNumeroInvitati() {
      return numeroInvitati;
    }

    public int inserisciInvitato(Invitato persona)
                throws TabellaException {
      String chiave = persona.getNome();
```

```
      int p = cercaInvitato(chiave, 'I');

      festa[p] = new Invitato(persona);
      numeroInvitati++;
      return p;
    }

    public int eliminaInvitato(String nome) throws TabellaException {
      int p = cercaInvitato(nome, 'R');

      festa[p] = null;
      numeroInvitati--;
      return p;
    }

    // funzione di visualizzazione dello stato corrente
    // utilizzata a scopo di test
    public void visualizzaStatoTabella() {
      for (int i=0; i<festa.length; i++)
        if (festa[i] != null)
          System.out.println("Posizione "+i+"->"+festa[i].toString()+
                             "(hash: "+hash(festa[i].getNome())+")");
    }

  private int scansioneLineare (int posizioneIniziale, String chiave,
                                char operazione)
            throws TabellaException {
    int p = posizioneIniziale;

    while (true) {
      p++;
      p = p % festa.length;
      if (operazione == 'R') {
        // ricerca per informazione
        if (festa[p] == null)
          throw new TabellaException("Elemento "+chiave+ " non esistente");
          if (festa[p].getNome().equals(chiave))
            return p;
      }
      else {
        // ricerca per inserimento
        if (festa[p] == null)
          return p;
        if (festa[p].getNome().equals(chiave))
          throw new TabellaException("Elemento "+festa[p].getNome()+ " preesistente");
      }
        if (p == posizioneIniziale)
          if (operazione == 'R')
            throw new TabellaException("Elemento "+chiave+ " non esistente");
          else
            throw new TabellaException("Tabella piena");
    }
  }
```

Il seguente *main* che ha lo scopo di testare i metodi della classe *TabellaInvitati*:

```
  public static void main (String[] args) {
    Invitato i1 = new Invitato("Bianchi Giovanni",'M',"0586 854822");
    Invitato i2 = new Invitato("Rossi Marta",'F',"0586 844853")
```

```
  Invitato i3 = new Invitato("Neri Marco",'M',"0586 444722");
  Invitato i4 = new Invitato("Verdi Roberta",'F',"0586 974824");
  Invitato i5 = new Invitato("Marti Rossa",'F',"0586 864253");

  TabellaInvitati t = new TabellaInvitati(100);

  try {
    System.out.println("Inserimento invitato " +i1.getNome()+
                       " posizione: "+t.inserisciInvitato(i1));
    System.out.println("Inserimento invitato " +i2.getNome()+
                       " posizione: "+t.inserisciInvitato(i2));
    System.out.println("Inserimento invitato " +i3.getNome()+
                       " posizione: "+t.inserisciInvitato(i3));
    System.out.println("Inserimento invitato " +i4.getNome()+
                       " posizione: "+t.inserisciInvitato(i4));
    System.out.println("Inserimento invitato " +i5.getNome()+
                       " posizione: "+t.inserisciInvitato(i5));
    System.out.println("Inserimento invitato " +i5.getNome()+
                       " posizione: "+t.inserisciInvitato(i5));
  }
  catch (TabellaException e) {
    System.out.println(e.getError());
  }

  System.out.println("Stato tabella:");
  t.visualizzaStatoTabella();

  try {
    System.out.println("Ricerca Marti Rossa: "+ t.cerca("Marti Rossa"));
    System.out.println("Ricerca Pippicalzelunghe: "+ t.cerca("Pippicalzelunghe"));
  }
  catch (TabellaException e) {
    System.out.println(e.getError());
  }

try {
  System.out.println("Eliminazione invitato Verdi Roberta in posizione: "+
                     t.eliminaInvitato("Verdi Roberta"));
  System.out.println("Eliminazione invitato Marti Rossa in posizione: "+
                     t.eliminaInvitato("Marti Rossa"));
  System.out.println("Eliminazione invitato Verdi Roberta in posizione: "+
                     t.eliminaInvitato("Verdi Roberta"));
}
catch (TabellaException e) {
  System.out.println(e.getError());
}

  System.out.println("Stato tabella:");
  t.visualizzaStatoTabella();
}
```

Produce in output:

```
Inserimento invitato Bianchi Giovanni posizione: 45
Inserimento invitato Rossi Marta posizione: 61
Inserimento invitato Neri Marco posizione: 28
Inserimento invitato Verdi Roberta posizione: 57
Inserimento invitato Marti Rossa posizione: 62
Elemento Marti Rossa preesistente
```

```
  Invitato i3 = new Invitato("Neri Marco",'M',"0586 444722");
  Invitato i4 = new Invitato("Verdi Roberta",'F',"0586 974824");
  Invitato i5 = new Invitato("Marti Rossa",'F',"0586 864253");

  TabellaInvitati t = new TabellaInvitati(100);

  try {
    System.out.println("Inserimento invitato " +i1.getNome()+
                       " posizione: "+t.inserisciInvitato(i1));
    System.out.println("Inserimento invitato " +i2.getNome()+
                       " posizione: "+t.inserisciInvitato(i2));
    System.out.println("Inserimento invitato " +i3.getNome()+
                       " posizione: "+t.inserisciInvitato(i3));
    System.out.println("Inserimento invitato " +i4.getNome()+
                       " posizione: "+t.inserisciInvitato(i4));
    System.out.println("Inserimento invitato " +i5.getNome()+
                       " posizione: "+t.inserisciInvitato(i5));
    System.out.println("Inserimento invitato " +i5.getNome()+
                       " posizione: "+t.inserisciInvitato(i5));
  }
  catch (TabellaException e) {
    System.out.println(e.getError());
  }

  System.out.println("Stato tabella:");
  t.visualizzaStatoTabella();

  try {
    System.out.println("Ricerca Marti Rossa: "+ t.cerca("Marti Rossa"));
    System.out.println("Ricerca Pippicalzelunghe: "+ t.cerca("Pippicalzelunghe"));
  }
  catch (TabellaException e) {
    System.out.println(e.getError());
  }

try {
  System.out.println("Eliminazione invitato Verdi Roberta in posizione: "+
                     t.eliminaInvitato("Verdi Roberta"));
  System.out.println("Eliminazione invitato Marti Rossa in posizione: "+
                     t.eliminaInvitato("Marti Rossa"));
  System.out.println("Eliminazione invitato Verdi Roberta in posizione: "+
                     t.eliminaInvitato("Verdi Roberta"));
}
catch (TabellaException e) {
  System.out.println(e.getError());
}

  System.out.println("Stato tabella:");
  t.visualizzaStatoTabella();
}
```

Produce in output:

```
Inserimento invitato Bianchi Giovanni posizione: 45
Inserimento invitato Rossi Marta posizione: 61
Inserimento invitato Neri Marco posizione: 28
Inserimento invitato Verdi Roberta posizione: 57
Inserimento invitato Marti Rossa posizione: 62
Elemento Marti Rossa preesistente
```

```
Stato tabella:
Posizione 28 -> Neri Marco M 0586 444722(hash: 28)
Posizione 45 -> Bianchi Giovanni M 0586 854822(hash: 45)
Posizione 57 -> Verdi Roberta F 0586 974824(hash: 57)
Posizione 61 -> Rossi Marta F 0586 844853(hash: 61)
Posizione 62 -> Marti Rossa F 0586 864253(hash: 61)
Ricerca Marti Rossa: Marti Rossa F 0586 864253
Elemento Pippicalzelunghe non esistente
Eliminazione invitato Verdi Roberta in posizione: 57
Eliminazione invitato Marti Rossa in posizione: 62
Elemento Verdi Roberta non esistente
Stato tabella:
Posizione 28 -> Neri Marco M 0586 444722(hash: 28)
Posizione 45 -> Bianchi Giovanni M 0586 854822(hash: 45)
Posizione 61 -> Rossi Marta F 0586 844853(hash: 61)
```

Si noti nell'esempio precedente come l'inserimento del nome «Marti Rossa» abbia generato una collisione nella posizione 61 dove era memorizzato l'elemento di chiave «Rossi Marta» (non poteva essere altrimenti essendo le due chiavi una l'anagramma dell'altra): l'algoritmo di scansione lineare di passo unitario ha trovato il primo elemento disponibile alla posizione successiva.

La TABELLA 2 confronta vantaggi e svantaggi dell'albero binario di ricerca con la tabella a indirizzamento *hash* per la gestione di informazioni.

TABELLA 2

	Vantaggi	Svantaggi
Bst	• Occupazione di memoria limitata al numero di elementi effettivamente memorizzati. • Gestione dinamica del numero di elementi. • Ordinamento predefinito degli elementi (in funzione di un loro valore chiave).	• Necessità di bilanciare l'albero a ogni inserimento/eliminazione di un elemento per mantenerne le prestazioni in fase di ricerca.
Hash	• Rapidità delle operazioni di ricerca e inserimento/eliminazione di elementi.	• Numero massimo prefissato di elementi. • Occupazione di memoria pari al numero massimo di elementi e maggiore delle effettive necessità. • Nessun ordinamento predefinito: per ottenere le informazioni ordinate secondo un qualunque criterio è necessario applicare un algoritmo di *sort* su una copia esterna delle chiavi della tabella.

```
Stato tabella:
Posizione 28 -> Neri Marco M 0586 444722(hash: 28)
Posizione 45 -> Bianchi Giovanni M 0586 854822(hash: 45)
Posizione 57 -> Verdi Roberta F 0586 974824(hash: 57)
Posizione 61 -> Rossi Marta F 0586 844853(hash: 61)
Posizione 62 -> Marti Rossa F 0586 864253(hash: 61)
Ricerca Marti Rossa: Marti Rossa F 0586 864253
Elemento Pippicalzelunghe non esistente
Eliminazione invitato Verdi Roberta in posizione: 57
Eliminazione invitato Marti Rossa in posizione: 62
Elemento Verdi Roberta non esistente
Stato tabella:
Posizione 28 -> Neri Marco M 0586 444722(hash: 28)
Posizione 45 -> Bianchi Giovanni M 0586 854822(hash: 45)
Posizione 61 -> Rossi Marta F 0586 844853(hash: 61)
```

Si noti nell'esempio precedente come l'inserimento del nome «Marti Rossa» abbia generato una collisione nella posizione 61 dove era memorizzato l'elemento di chiave «Rossi Marta» (non poteva essere altrimenti essendo le due chiavi una l'anagramma dell'altra): l'algoritmo di scansione lineare di passo unitario ha trovato il primo elemento disponibile alla posizione successiva.

La TABELLA 2 confronta vantaggi e svantaggi dell'albero binario di ricerca con la tabella a indirizzamento *hash* per la gestione di informazioni.

TABELLA 2

	Vantaggi	Svantaggi
Bst	• Occupazione di memoria limitata al numero di elementi effettivamente memorizzati. • Gestione dinamica del numero di elementi. • Ordinamento predefinito degli elementi (in funzione di un loro valore chiave).	• Necessità di bilanciare l'albero a ogni inserimento/eliminazione di un elemento per mantenerne le prestazioni in fase di ricerca.
Hash	• Rapidità delle operazioni di ricerca e inserimento/eliminazione di elementi.	• Numero massimo prefissato di elementi. • Occupazione di memoria pari al numero massimo di elementi e maggiore delle effettive necessità. • Nessun ordinamento predefinito: per ottenere le informazioni ordinate secondo un qualunque criterio è necessario applicare un algoritmo di *sort* su una copia esterna delle chiavi della tabella.

I CONCETTI CHIAVE

LISTA. È un insieme di elementi (nodi) collegati sequenzialmente tra loro tramite riferimenti (*link*); è una struttura dati dinamica in quanto è possibile aggiungere o eliminare singoli elementi seconda le necessità. Il riferimento iniziale (*head*) permette l'accesso al primo nodo della lista; l'ultimo nodo della lista ha il riferimento al nodo successivo nullo. L'accesso agli elementi di una lista è strettamente sequenziale.

NODO. È l'elemento costitutivo di una lista; la sua struttura può essere suddivisa logicamente in due componenti: una componente informativa e una componente di collegamento al nodo successivo (*link*). La componente informativa è un riferimento a un oggetto; la componente di collegamento è un riferimento a un oggetto di tipo nodo.

INSERIMENTO/ELIMINAZIONE NODI. In una lista è possibile aggiungere ed eliminare nodi in funzione delle necessità: queste operazioni avvengono operando opportunamente sui riferimenti (*link*) dei nodi della lista. In generale è necessario, per entrambi i tipi di operazione, distinguere il punto in cui si deve intervenire: in testa (è necessario modificare il riferimento iniziale *head*), in coda (è necessario modificare il riferimento di fine lista), in un punto intermedio (è necessario modificare il riferimento dell'elemento che precede il punto in cui si intende operare).

LISTA MULTIPLA. È una lista i cui nodi possono avere più di un componente di riferimento di altri nodi, in modo tale da poter costruire strutture dinamiche complesse dotate di vari livelli di sottoliste.

ITERATORE. Dato un oggetto aggregato (cioè un oggetto che contiene altri oggetti per fornire di questi una visione unitaria) un Iteratore è un oggetto che rende disponibili metodi per accedere sequenzialmente ai singoli elementi dell'oggetto aggregato senza esporne la rappresentazione interna. Si tratta di un *pattern* di progettazione standard.

CLASSI *NESTED*. Le classi Java *nested* sono classi contenute in altre classi. Esse si dividono in *static* e *non-static* (o *inner classes*) secondo la seguente articolazione: *static class* (membri statici di altre classi), *inner class* (membri non statici di altre classi), *local inner class* (dichiarate all'interno dei metodi di altre classi), *anonymous inner class* (sono simili alle precedenti ma scritte come un'espressione che ritorna un oggetto).

PILA. La pila (*stack*) è una struttura dati che adotta una politica di tipo LIFO (*Last-In First-Out*). Le pile sono generalmente implementate tramite liste in cui le operazioni di inserimento (*push*) ed estrazione (*pop*) avvengono sempre in testa.

CODA. La coda (*queue*) è una struttura dati che adotta una politica di tipo FIFO (*First-In First-Out*). Le code sono generalmente implementate tramite liste in cui le operazioni di inserimento (*enqueue*) vengono effettuate in coda, mentre quelle di estrazione (*dequeue*) avvengono in testa.

ALBERO GENERICO. Un albero è un insieme non vuoto di *nodi* (elementi aventi valenza informativa) e di *archi* (elementi aventi valenza connettiva che realizzano i collegamenti tra coppie di nodi), tale che sia: *connesso*, cioè presi due nodi qualsiasi esista un percorso semplice tra di essi; *aciclico*, ovvero non esista alcun percorso semplice in cui il nodo iniziale e quello finale siano coincidenti. Un albero con n nodi ha $n - 1$ archi; esiste inoltre un solo percorso semplice tra ogni coppia di nodi di un albero. Un albero è definito *albero con radice* se uno dei nodi dell'albero viene scelto come nodo radice rispetto al quale l'albero diviene una struttura gerarchica in cui i nodi sono ordinati rispetto ai nodi contigui in *padri* e *figli* (nodi che sono figli dello stesso padre sono *fratelli*).

MEMORIZZAZIONE DI UN ALBERO. Una possibile tecnica di memorizzazione di un albero prevede l'utilizzazione di una lista multipla i cui nodi (che corrispondono ai nodi dell'albero) sono strutturati in tre componenti: la componente informativa, il riferimento alla lista dei figli e il riferimento alla lista dei fratelli.

ORDINE DI VISITA DI UN ALBERO. I criteri di visita classici per un albero sono l'*ordine anticipato* e l'*ordine differito*. Nel primo caso si visita prima la radice e poi ricorsivamente i singoli sottoalberi in ordine anticipato; nel secondo caso la radice dell'albero e, ricorsivamente, dei suoi sottoalberi viene visitata per ultima.

ALBERO BINARIO. Un albero binario (BT, *Binary Tree*) è un albero in cui ogni nodo ha al massimo due figli, denominati rispettivamente *figlio sinistro* e *figlio destro*.

MEMORIZZAZIONE DI UN ALBERO BINARIO. Un albero binario può essere memorizzato tramite una lista multipla i cui nodi (che corrispondono ai nodi dell'albero) sono strutturati in tre componenti: la componente informativa, il riferimento al nodo radice del sottoalbero sinistro e il riferimento al nodo radice del sottoalbero destro.

ORDINE DI VISITA DI UN ALBERO BINARIO. Oltre ai metodi di visita anticipato e differito, per un albero binario è definito un terzo metodo di visita: l'ordine di visita *simmetrico* prevede che per ogni nodo dell'albero si visiti prima il sottoalbero sinistro, poi la radice e infine il sottoalbero destro.

I CONCETTI CHIAVE

LISTA. È un insieme di elementi (nodi) collegati sequenzialmente tra loro tramite riferimenti (*link*); è una struttura dati dinamica in quanto è possibile aggiungere o eliminare singoli elementi seconda le necessità. Il riferimento iniziale (*head*) permette l'accesso al primo nodo della lista; l'ultimo nodo della lista ha il riferimento al nodo successivo nullo. L'accesso agli elementi di una lista è strettamente sequenziale.

NODO. È l'elemento costitutivo di una lista; la sua struttura può essere suddivisa logicamente in due componenti: una componente informativa e una componente di collegamento al nodo successivo (*link*). La componente informativa è un riferimento a un oggetto; la componente di collegamento è un riferimento a un oggetto di tipo nodo.

INSERIMENTO/ELIMINAZIONE NODI. In una lista è possibile aggiungere ed eliminare nodi in funzione delle necessità: queste operazioni avvengono operando opportunamente sui riferimenti (*link*) dei nodi della lista. In generale è necessario, per entrambi i tipi di operazione, distinguere il punto in cui si deve intervenire: in testa (è necessario modificare il riferimento iniziale *head*), in coda (è necessario modificare il riferimento di fine lista), in un punto intermedio (è necessario modificare il riferimento dell'elemento che precede il punto in cui si intende operare).

LISTA MULTIPLA. È una lista i cui nodi possono avere più di un componente di riferimento di altri nodi, in modo tale da poter costruire strutture dinamiche complesse dotate di vari livelli di sottoliste.

ITERATORE. Dato un oggetto aggregato (cioè un oggetto che contiene altri oggetti per fornire di questi una visione unitaria) un Iteratore è un oggetto che rende disponibili metodi per accedere sequenzialmente ai singoli elementi dell'oggetto aggregato senza esporne la rappresentazione interna. Si tratta di un *pattern* di progettazione standard.

CLASSI *NESTED*. Le classi Java *nested* sono classi contenute in altre classi. Esse si dividono in *static* e *non-static* (o *inner classes*) secondo la seguente articolazione: *static class* (membri statici di altre classi), *inner class* (membri non statici di altre classi), *local inner class* (dichiarate all'interno dei metodi di altre classi), *anonymous inner class* (sono simili alle precedenti ma scritte come un'espressione che ritorna un oggetto).

PILA. La pila (*stack*) è una struttura dati che adotta una politica di tipo LIFO (*Last-In First-Out*). Le pile sono generalmente implementate tramite liste in cui le operazioni di inserimento (*push*) ed estrazione (*pop*) avvengono sempre in testa.

CODA. La coda (*queue*) è una struttura dati che adotta una politica di tipo FIFO (*First-In First-Out*). Le code sono generalmente implementate tramite liste in cui le operazioni di inserimento (*enqueue*) vengono effettuate in coda, mentre quelle di estrazione (*dequeue*) avvengono in testa.

ALBERO GENERICO. Un albero è un insieme non vuoto di *nodi* (elementi aventi valenza informativa) e di *archi* (elementi aventi valenza connettiva che realizzano i collegamenti tra coppie di nodi), tale che sia: *connesso*, cioè presi due nodi qualsiasi esista un percorso semplice tra di essi; *aciclico*, ovvero non esista alcun percorso semplice in cui il nodo iniziale e quello finale siano coincidenti. Un albero con n nodi ha $n - 1$ archi; esiste inoltre un solo percorso semplice tra ogni coppia di nodi di un albero. Un albero è definito *albero con radice* se uno dei nodi dell'albero viene scelto come nodo radice rispetto al quale l'albero diviene una struttura gerarchica in cui i nodi sono ordinati rispetto ai nodi contigui in *padri* e *figli* (nodi che sono figli dello stesso padre sono *fratelli*).

MEMORIZZAZIONE DI UN ALBERO. Una possibile tecnica di memorizzazione di un albero prevede l'utilizzazione di una lista multipla i cui nodi (che corrispondono ai nodi dell'albero) sono strutturati in tre componenti: la componente informativa, il riferimento alla lista dei figli e il riferimento alla lista dei fratelli.

ORDINE DI VISITA DI UN ALBERO. I criteri di visita classici per un albero sono l'*ordine anticipato* e l'*ordine differito*. Nel primo caso si visita prima la radice e poi ricorsivamente i singoli sottoalberi in ordine anticipato; nel secondo caso la radice dell'albero e, ricorsivamente, dei suoi sottoalberi viene visitata per ultima.

ALBERO BINARIO. Un albero binario (BT, *Binary Tree*) è un albero in cui ogni nodo ha al massimo due figli, denominati rispettivamente *figlio sinistro* e *figlio destro*.

MEMORIZZAZIONE DI UN ALBERO BINARIO. Un albero binario può essere memorizzato tramite una lista multipla i cui nodi (che corrispondono ai nodi dell'albero) sono strutturati in tre componenti: la componente informativa, il riferimento al nodo radice del sottoalbero sinistro e il riferimento al nodo radice del sottoalbero destro.

ORDINE DI VISITA DI UN ALBERO BINARIO. Oltre ai metodi di visita anticipato e differito, per un albero binario è definito un terzo metodo di visita: l'ordine di visita *simmetrico* prevede che per ogni nodo dell'albero si visiti prima il sottoalbero sinistro, poi la radice e infine il sottoalbero destro.

ALBERO BINARIO DI RICERCA. Un albero binario viene detto albero binario di ricerca (BST, *Binary Search Tree*) se per ogni nodo *n* il sottoalbero di sinistra contiene valori minori (secondo un ordinamento prefissato) del valore contenuto in *n*, e il sottoalbero di destra contiene valori maggiori di questo. Visitando in ordine simmetrico un albero binario di ricerca si ottiene una sequenza di nodi in accordo con il criterio di ordinamento prefissato.

ALBERO BINARIO DI RICERCA BILANCIATO. Un albero binario di ricerca è bilanciato se per l'albero stesso e ogni sottoalbero il numero di nodi presenti nel sottoalbero sinistro differisce al più di 1 dal numero di nodi del sottoalbero destro. In un albero binario di ricerca bilanciato con *n* nodi la ricerca di uno specifico nodo richiede al massimo di visitare $\log_2 n$ nodi.

TABELLE. Una tabella è una successione di elementi informativi, ciascuno dei quali composto da due parti logicamente distinte: la *chiave k* e l'*informazione i* a essa associata. Le tabelle possono essere implementate nel linguaggio di programmazione Java come *array* di oggetti. L'operazione fondamentale che si applica a una tabella è la *ricerca* di un elemento a partire dal valore della chiave.

INDIRIZZAMENTO *HASH*. È una tecnica basata sull'applicazione di un'opportuna funzione (funzione *hash*) alla chiave per calcolarne direttamente la posizione all'interno della tabella. La validità di una funzione *hash* è data da un insieme di fattori, tra cui il *basso costo*, il *determinismo* computazionale e l'*uniformità* della distribuzione dei valori calcolati. La stessa funzione *hash* deve essere utilizzata sia per memorizzare le chiavi nella tabella sia per ricercarle successivamente.

COLLISIONI. Si ha una collisione quando una funzione *hash* applicata a chiavi diverse genera lo stesso indirizzo. Una gestione efficiente delle collisioni è fondamentale per l'impiego di un indirizzamento *hash*. Esistono due tipologie di metodi per risolvere il problema delle collisioni: *concatenamento* e *indirizzamento aperto*. Nel primo caso ogni posizione della tabella è una lista in cui sono memorizzate tutte le chiavi che collidono nella posizione; nel secondo caso tutte le chiavi sono contenute nella tabella: se l'inserimento di una nuova chiave genera una collisione, essa viene memorizzata in una diversa posizione libera secondo una legge predefinita (nel caso della scansione lineare di passo 1 si tratta della prima posizione libera successiva).

ALBERO BINARIO DI RICERCA. Un albero binario viene detto albero binario di ricerca (BST, *Binary Search Tree*) se per ogni nodo *n* il sottoalbero di sinistra contiene valori minori (secondo un ordinamento prefissato) del valore contenuto in *n*, e il sottoalbero di destra contiene valori maggiori di questo. Visitando in ordine simmetrico un albero binario di ricerca si ottiene una sequenza di nodi in accordo con il criterio di ordinamento prefissato.

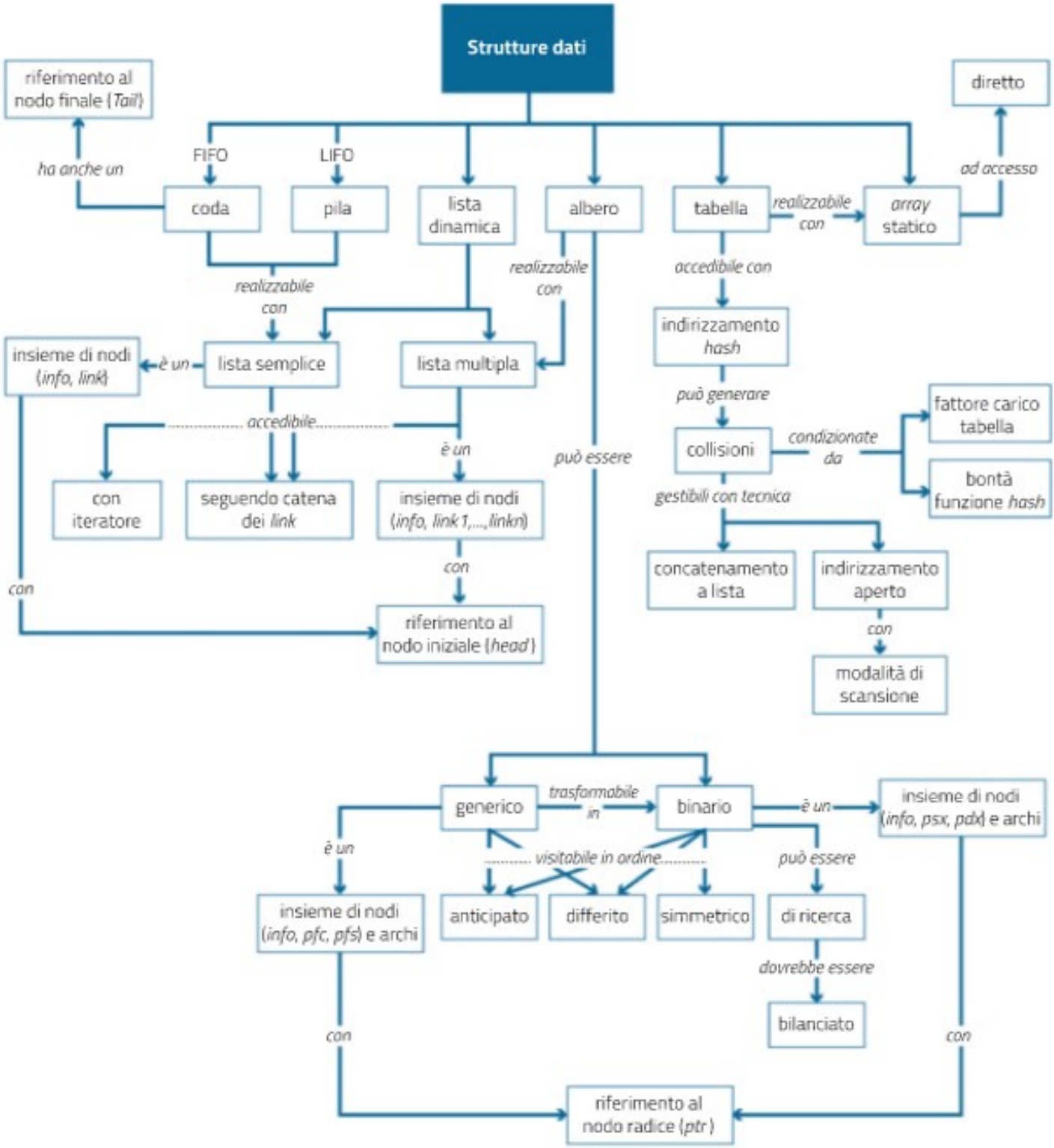
ALBERO BINARIO DI RICERCA BILANCIATO. Un albero binario di ricerca è bilanciato se per l'albero stesso e ogni sottoalbero il numero di nodi presenti nel sottoalbero sinistro differisce al più di 1 dal numero di nodi del sottoalbero destro. In un albero binario di ricerca bilanciato con *n* nodi la ricerca di uno specifico nodo richiede al massimo di visitare $\log_2 n$ nodi.

TABELLE. Una tabella è una successione di elementi informativi, ciascuno dei quali composto da due parti logicamente distinte: la *chiave k* e l'*informazione i* a essa associata. Le tabelle possono essere implementate nel linguaggio di programmazione Java come *array* di oggetti. L'operazione fondamentale che si applica a una tabella è la *ricerca* di un elemento a partire dal valore della chiave.

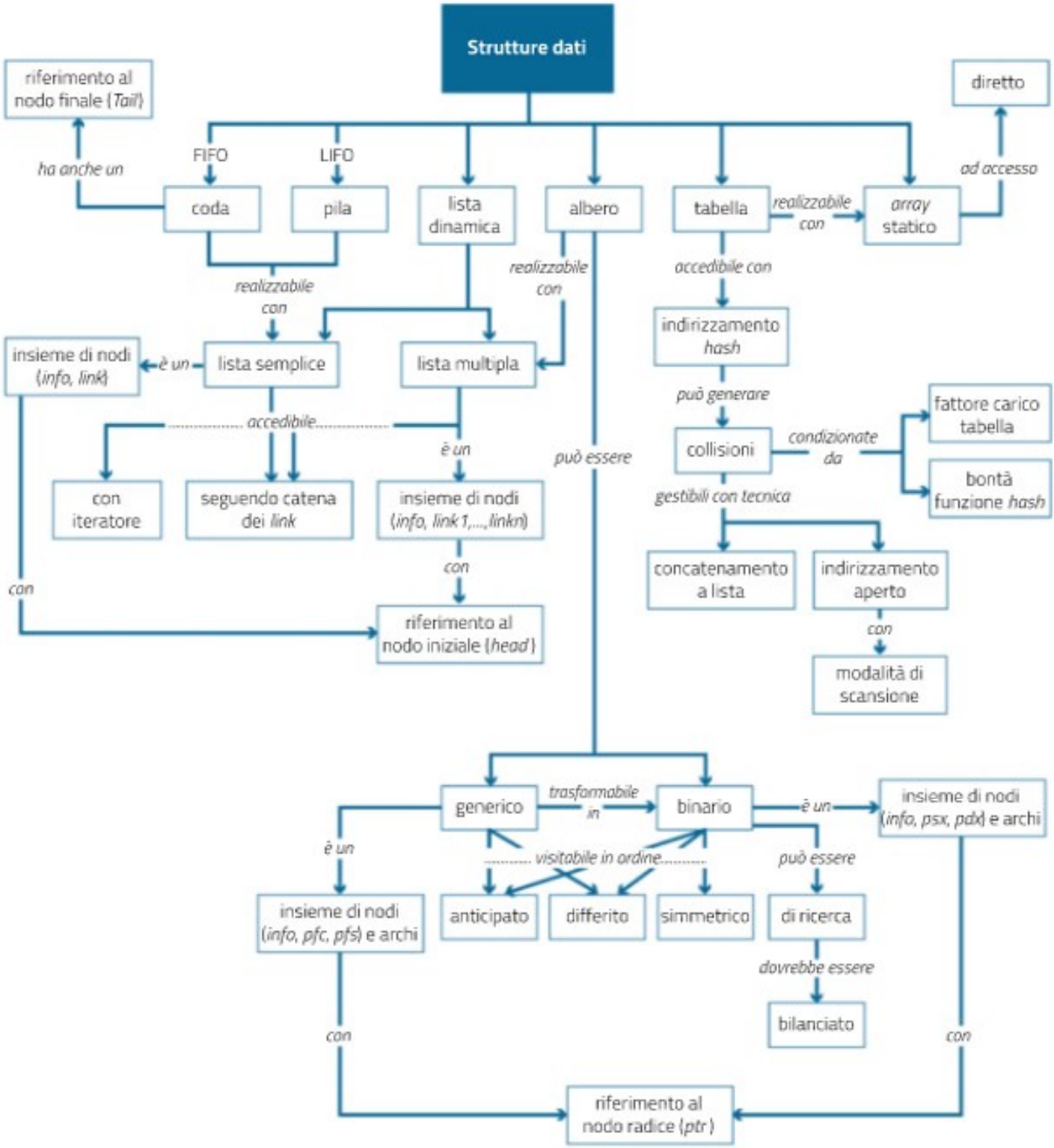
INDIRIZZAMENTO *HASH*. È una tecnica basata sull'applicazione di un'opportuna funzione (funzione *hash*) alla chiave per calcolarne direttamente la posizione all'interno della tabella. La validità di una funzione *hash* è data da un insieme di fattori, tra cui il *basso costo*, il *determinismo* computazionale e l'*uniformità* della distribuzione dei valori calcolati. La stessa funzione *hash* deve essere utilizzata sia per memorizzare le chiavi nella tabella sia per ricercarle successivamente.

COLLISIONI. Si ha una collisione quando una funzione *hash* applicata a chiavi diverse genera lo stesso indirizzo. Una gestione efficiente delle collisioni è fondamentale per l'impiego di un indirizzamento *hash*. Esistono due tipologie di metodi per risolvere il problema delle collisioni: *concatenamento* e *indirizzamento aperto*. Nel primo caso ogni posizione della tabella è una lista in cui sono memorizzate tutte le chiavi che collidono nella posizione; nel secondo caso tutte le chiavi sono contenute nella tabella: se l'inserimento di una nuova chiave genera una collisione, essa viene memorizzata in una diversa posizione libera secondo una legge predefinita (nel caso della scansione lineare di passo 1 si tratta della prima posizione libera successiva).

RIPASSA CON LA MAPPA



RIPASSA CON LA MAPPA



SPIEGA IN UNA PAGINA

1 Descrivere le caratteristiche base di una lista semplice e le classi necessarie per una sua implementazione in linguaggio Java.

2 Descrivere il meccanismo dell'indirizzamento *hash* e il problema delle collisioni.

QUESITI A RISPOSTA APERTA

3 Quali sono e come funzionano i metodi base di visita di un albero binario?

4 Quali sono le caratteristiche peculiari delle due strutture dati *Pila* e *Coda*?

5 Da dove nasce la necessità di mantenere bilanciato un albero binario di ricerca?

6 Che cos'è il fattore di carico di una tabella *hash* e a quali caratteristiche dovrebbe rispondere?

7 Che cosa sono le classi *nested*?

8 Indicare lo scopo del seguente metodo Java di un'ipotetica classe *Lista* specificando il significato del valore restituito.

```
public int mistery(int z)
  {
    Nodo p = head;
    if (p==null)
      return -1;
    while ((p.getLink()!=0) &&
           (!p.getInfo() == z))
        p=p.getLink();
    if (p.getInfo() == z)
      return 0;
    Nodo pn = new Nodo(z,null);
    p.setLink(pn);
    return 1;
}
```

9 Visitando un albero binario in ordine differito si ottiene la sequenza: F, D, H, G, P, W, Z, T, R, I. Lo stesso albero, visitato in ordine simmetrico, fornisce la sequenza D, F, G, H, I, P, R, T, W, Z. Ricostruire l'albero binario specificando se si tratta o meno di un albero binario di ricerca.

10 Che cosa restituisce il seguente metodo Java di un'ipotetica classe *AlberoBinario* in cui il contenuto informativo di nodi sono valori interi?

```
private int mistery(Nodo n) {
  int x;

  if (n == null) return 0;
  x = mistery(n.getRightChild());
  x += mistery(n.getLeftChild());
  x += n.getInfo();
  return x;
}

public int mistery() {
  return mistery(root);
}
```

TEST A RISPOSTA MULTIPLA

11 Una lista è una struttura dati ad accesso:

- A diretto.
- B sequenziale.
- C sia diretto sia sequenziale.
- D inverso.

12 Il numero degli elementi di una lista è:

- A prefissato.
- B variabile.
- C maggiore di 1.
- D minore di un valore prefissato.

13 Porre uguale a `null` il riferimento *head* di una lista equivale a:

- A perdere il primo nodo della lista.
- B perdere l'ultimo nodo della lista.
- C perdere tutti i nodi della lista.
- D eliminare completamente l'oggetto *lista*.

14 Perché si possa utilizzare una tecnica di indirizzamento *hash* è necessario che la struttura a cui viene applicata sia:

- A ordinata.
- B priva di chiavi duplicate.
- C ad accesso sequenziale.
- D ad accesso diretto.

15 In un'operazione di inserimento in testa di un nodo in una lista è necessario:

- A aggiornare comunque il riferimento di fine lista.
- B aggiornare comunque il riferimento *head*.

SPIEGA IN UNA PAGINA

1 Descrivere le caratteristiche base di una lista semplice e le classi necessarie per una sua implementazione in linguaggio Java.

2 Descrivere il meccanismo dell'indirizzamento *hash* e il problema delle collisioni.

QUESITI A RISPOSTA APERTA

3 Quali sono e come funzionano i metodi base di visita di un albero binario?

4 Quali sono le caratteristiche peculiari delle due strutture dati *Pila* e *Coda*?

5 Da dove nasce la necessità di mantenere bilanciato un albero binario di ricerca?

6 Che cos'è il fattore di carico di una tabella *hash* e a quali caratteristiche dovrebbe rispondere?

7 Che cosa sono le classi *nested*?

8 Indicare lo scopo del seguente metodo Java di un'ipotetica classe *Lista* specificando il significato del valore restituito.

```
public int mistery(int z)
  {
    Nodo p = head;
    if (p==null)
      return -1;
    while ((p.getLink()!=0) &&
           (!p.getInfo() == z))
        p=p.getLink();
    if (p.getInfo() == z)
      return 0;
    Nodo pn = new Nodo(z,null);
    p.setLink(pn);
    return 1;
}
```

9 Visitando un albero binario in ordine differito si ottiene la sequenza: F, D, H, G, P, W, Z, T, R, I. Lo stesso albero, visitato in ordine simmetrico, fornisce la sequenza D, F, G, H, I, P, R, T, W, Z. Ricostruire l'albero binario specificando se si tratta o meno di un albero binario di ricerca.

10 Che cosa restituisce il seguente metodo Java di un'ipotetica classe *AlberoBinario* in cui il contenuto informativo di nodi sono valori interi?

```
private int mistery(Nodo n) {
  int x;

  if (n == null) return 0;
  x = mistery(n.getRightChild());
  x += mistery(n.getLeftChild());
  x += n.getInfo();
  return x;
}

public int mistery() {
  return mistery(root);
}
```

TEST A RISPOSTA MULTIPLA

11 Una lista è una struttura dati ad accesso:

- A diretto.
- B sequenziale.
- C sia diretto sia sequenziale.
- D inverso.

12 Il numero degli elementi di una lista è:

- A prefissato.
- B variabile.
- C maggiore di 1.
- D minore di un valore prefissato.

13 Porre uguale a `null` il riferimento *head* di una lista equivale a:

- A perdere il primo nodo della lista.
- B perdere l'ultimo nodo della lista.
- C perdere tutti i nodi della lista.
- D eliminare completamente l'oggetto *lista*.

14 Perché si possa utilizzare una tecnica di indirizzamento *hash* è necessario che la struttura a cui viene applicata sia:

- A ordinata.
- B priva di chiavi duplicate.
- C ad accesso sequenziale.
- D ad accesso diretto.

15 In un'operazione di inserimento in testa di un nodo in una lista è necessario:

- A aggiornare comunque il riferimento di fine lista.
- B aggiornare comunque il riferimento *head*.

C Non occorre aggiornare alcun riferimento.
D aggiornare i riferimenti di tutti i nodi della lista.

16 In una lista multipla ogni nodo deve avere:

A almeno due componenti con valenza di riferimento (*link*) distinti.
B almeno due componenti con valenza informativa distinti.
C più di due componenti con valenza di riferimento (*link*) distinti.
D più di un riferimento *head*.

17 Per la memorizzazione di un albero è:

A sufficiente una lista semplice.
B necessaria una lista multipla.
C necessario un *array* multidimensionale.
D preferibile una tabella a indirizzamento *hash*.

18 Per realizzare un algoritmo iterativo di visita anticipata di un albero è necessario:

A utilizzare una coda per i riferimenti ai nodi.
B effettuare preventivamente un ordinamento dei nodi dell'albero.
C avere una pila per i riferimenti ai nodi.
D È possibile visitare un albero solo con un algoritmo ricorsivo.

19 Applicando un algoritmo di visita differita a un albero, l'ultimo nodo visitato è:

A la radice dell'albero stesso.
B il primo figlio della radice.
C l'ultimo nodo del sottoalbero che ha come radice l'ultimo figlio della radice.
D il nodo con profondità maggiore.

20 Applicando un algoritmo di visita simmetrica a un albero binario, l'ultimo nodo visitato è:

A la radice dell'albero stesso.
B il primo figlio di sinistra della radice.
C il nodo con maggiore profondità del sottoalbero destro della radice e, a parità di profondità, quello che comunque si trova a destra.
D il nodo con profondità maggiore.

21 La politica LIFO è propria di:

A un *array* monodimensionale.
B una tabella.
C una pila.
D una coda.

22 Data una funzione *hash h* e una qualsiasi coppia di chiavi k_1 e k_2, il fatto che $h(k_1) \neq h(k_2)$ è condizione:

A necessaria.
B indifferente.
C auspicabile.
D sufficiente.

23 È sempre possibile applicare una tecnica di indirizzamento *hash* a una lista semplice?

A Sì, sempre.
B No, mai.
C Dipende dal tipo di informazioni gestite nella lista.
D Dipende dal numero massimo di nodi della lista.

24 Data una funzione *hash h* applicata a una tabella e una chiave *k*, *h*(*k*) rappresenta:

A la posizione della chiave *k* nella tabella.
B la chiave *k* stessa.
C il fattore di carico della tabella.
D il numero di chiavi memorizzate nella tabella.

25 La politica FIFO è propria di:

A un *array* monodimensionale (vettore).
B una pila.
C una coda.
D una lista.

26 Quale algoritmo di visita di un albero binario di ricerca produce la sequenza ordinata del contenuto dei nodi?

A Visita in ordine anticipato.
B Visita in ordine differito.
C Visita in ordine simmetrico.
D Un qualsiasi tipo di visita.

PROBLEMI

Liste

27 Scrivere un metodo Java denominato *shuffle* per un'ipotetica classe *Lista* che, ricevendo come parametri due valori interi *k* e *h*, sposti l'elemento di posizione *k* nella posizione *h*. Il metodo deve restituire il valore 0 se $k = h$, il valore 1 se l'operazione ha avuto esito positivo, il valore −1 se l'operazione ha avuto esito negativo. Se $h \leq 1$ l'elemento deve essere posto nella prima posizione della lista, mentre se $h \geq n$, con *n* numero di elementi della lista, deve essere posto in ultima posizione.

28 Scrivere un metodo Java denominato *lastFirst* per un'ipotetica classe *Lista* che sposti in testa l'ultimo elemento e contemporaneamente sposti in coda il primo elemento.

29 Scrivere un costruttore per un'ipotetica classe Java *Lista* che inizializzi la lista con i valori contenuti in un vettore fornito come parametro. La memorizzazione deve avvenire in modo che i valori siano ordinati nella lista in senso inverso rispetto all'ordinamento originale del vettore.

30 Scrivere un metodo Java denominato *isSublist* per un'ipotetica classe *Lista* che, a partire da un oggetto istanza di *Lista* fornito come argomento, verifichi se esso è o meno una sottolista.

31 Scrivere un costruttore per un'ipotetica classe Java *Lista* che inizializzi la lista con la fusione ordinata degli elementi di due oggetti istanza di *Lista* forniti come argomento e i cui elementi sono originalmente ordinati in maniera crescente.

32 Scrivere un metodo Java denominato *erase* per un'ipotetica classe *Lista* che ricerchi e rimuova tutte le occorrenze nella lista dei nodi contenenti l'informazione fornita come parametro. Il metodo deve restituire il numero dei nodi rimossi.

33 In un dispositivo palmare abilitato alla ricezione della posta elettronica le e-mail sono mantenute in una lista ordinata in cui vengono inserite al momento della ricezione. Per ogni e-mail devono essere memorizzate le seguenti informazioni:

- mittente;
- oggetto;
- data e ora (si ipotizzi di disporre di una classe *Timestamp* già implementata);
- testo.

Progettare mediante un diagramma delle classi UML e implementare in linguaggio Java una classe *Mailbox* che consenta di eseguire le seguenti operazioni, gestendo in modo adeguato le relative eccezioni:

- aggiunta di un'e-mail alla lista tenendo conto che l'ultima ricevuta è sempre la prima della lista;
- eliminazione di un'e-mail data la sua posizione nella lista;
- ricerca di tutte le e-mail che contengono uno specifico testo nell'oggetto (la classe *String* dispone del metodo *contains* che restituisce un valore booleano a seconda che un oggetto di tipo *String* contenga o meno come sottostringa la stringa fornita come argomento).

34 A SCUOLA DI LAVORO Un centro di pronto soccorso registra i nominativi dei pazienti che necessitano di un intervento classificando i casi come «Rossi» (molto urgenti), «Gialli» (normalmente urgenti) e «Bianchi» (poco urgenti): ovviamente tutti i casi «Rossi» devono essere gestiti prima dei casi «Gialli» e tutti i casi «Gialli» prima di quelli «Bianchi» indipendentemente dall'ordine di arrivo. Progettare mediante un diagramma delle classi UML e implementare in linguaggio Java una classe *ProntoSoccorso* che consenta di eseguire le seguenti operazioni, gestendo in modo adeguato le relative eccezioni:

- aggiunta di un paziente alla lista di attesa in modo da rispettare l'ordine di priorità;
- recupero del nominativo ed eliminazione del paziente con il massimo grado di urgenza dalla lista;
- determinazione del numero di pazienti «Rossi», «Gialli» e «Bianchi» in attesa;
- eliminazione dalla lista di un paziente specifico.

Alberi

35 Scrivere un metodo Java per un'ipotetica classe *Albero* che, a partire dall'informazione di tipo stringa *k* fornita come parametro, restituisca il valore −1 se nessun nodo dell'albero contiene la stringa *k* o, in alternativa, il valore della profondità massima del sottoalbero che ha come radice il nodo che contiene la stringa *k*. Si supponga che l'albero sia implementato mediante una lista multipla.

36 Scrivere un metodo Java denominato *Boscaiolo* per un'ipotetica classe *AlberoBinario* che, a partire dall'informazione di tipo stringa *k* fornita

come parametro, rimuova ogni nodo dell'albero contenente la stringa *k* eliminando i relativi sottoalberi.

37 Scrivere un metodo Java *ricerca* per un'ipotetica classe *AlberoBinario* che determini la presenza o meno nei nodi dell'albero di uno specifico valore fornito come argomento.

38 Modificare la classe *NodoAlberoBinario* aggiungendo un attributo per il riferimento al nodo padre e i metodi necessari per la relativa gestione. Scrivere un metodo Java non ricorsivo per la visita di un albero binario i cui nodi sono di questo tipo.

Tabelle a indirizzamento *hash*

39 A SCUOLA DI LAVORO Una grande libreria intende gestire con un'applicazione Java i libri posseduti identificati dal codice ISBN e caratterizzati da vari attributi (autore, titolo, editore, anno di pubblicazione). Progettare mediante un diagramma UML delle classi e implementare in linguaggio Java una classe *Libreria* che memorizzi gli oggetti di tipo *Libro* in una tabella a indirizzamento *hash*. I metodi della classe devono consentire:

- la ricerca dei dati di un libro a partire dal proprio codice ISBN;
- la memorizzazione di un nuovo libro;
- l'eliminazione di un libro.

40 Un dispositivo portatile per l'acquisto dei prodotti in un supermercato deve mantenere l'elenco dei prodotti acquistati dal cliente che lo utilizza scansionando i codici a barre delle confezioni. Per ogni oggetto di classe *Prodotto* devono essere memorizzate le seguenti informazioni: codice a barre, descrizione, quantità, costo (come il dispositivo acquisisca le informazioni collegate al codice a barre è allo scopo ininfluente).
Progettare mediante un diagramma delle classi UML una classe *Spesa* che memorizzi i prodotti in una tabella a indirizzamento *hash* e consenta di eseguire le seguenti operazioni gestendo in modo adeguato le relative eccezioni:

- aggiunta di un prodotto all'elenco;
- eliminazione di un prodotto dall'elenco dato il codice a barre;
- calcolo del costo totale dei prodotti presenti nell'elenco.

41 Si intende realizzare una rubrica per un dispositivo palmare per gestire in via prioritaria i numeri telefonici dei contatti: progettare in notazione UML e implementare in linguaggio Java una classe per rappresentare un singolo contatto della rubrica (cognome, nome, indirizzo e-mail e numero di telefono). Quindi progettare in notazione UML e implementare in linguaggio Java una classe per realizzare la rubrica memorizzando i contatti in una tabella a indirizzamento *hash* che consenta di effettuare le seguenti operazioni:

- aggiunta di un nuovo contatto;
- ricerca delle informazioni di contatto a partire dal numero di telefono;
- eliminazione di un contatto specificando il numero di telefono;
- ricerca delle informazioni di contatto a partire dall'indirizzo e-mail;
- eliminazione di un contatto specificando l'indirizzo e-mail.

CASI PRATICI

42 Un dispositivo MP3 consente di gestire playlist di brani musicali descritti da un titolo e dalla durata espressa in secondi. Progettare mediante un diagramma delle classi UML e implementare in linguaggio Java le classi *Brano* e *Playlist* che consentano di eseguire le seguenti operazioni, gestendo in modo adeguato le relative eccezioni:

- aggiunta di un brano alla lista (se la lista comprende già altri brani l'inserimento deve avvenire nell'ultima posizione);
- eliminazione dalla lista di un brano identificato dal titolo;
- determinazione della durata totale dei brani della lista;
- esportazione dei primi *n* brani della lista in un vettore (con *n* fornito come parametro);
- esportazione dei primi brani della lista fino a un tempo complessivo *t* di riproduzione (con *t* fornito come parametro);
- spostamento di un brano identificato dalla posizione (2, 3, ..., *n*) nella posizione precedente (1, 2, ..., *n* – 1);
- spostamento di un brano identificato dalla posizione (1, 2, ..., *n* – 1) nella posizione successiva (2, 3, ..., *n*);
- salvataggio e ripristino della lista dei brani in/da un file di tipo testuale;
- riordino casuale dei brani della lista (funzio-

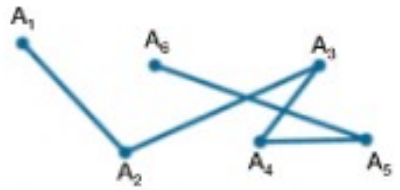
ne *shuffle*: il metodo *nextInt* della classe *java.util.Random* restituisce un numero casuale intero).

Realizzare un *main* che consenta all'utente di eseguire le operazioni in modo interattivo.

43 A SCUOLA DI LAVORO Un ufficio accetta le pratiche che deve svolgere da un sistema di immissione *online* accessibile dagli utenti mediante Internet: ogni pratica è caratterizzata da un numero progressivo assegnato automaticamente dal sistema, dal cognome e dal nome del richiedente, da un carattere che ne individua il tipo (A, B, C, ...) e da un breve testo di descrizione. Il sistema di gestione delle pratiche deve fare in modo che l'operatore prenda in carico sempre la prima pratica non evasa secondo una politica FIFO.
Si richiede un diagramma UML delle classi e una implementazione in linguaggio Java di una soluzione al problema descritto che utilizzi una lista in cui le pratiche sono inserite in testa e prelevate in coda (o viceversa); le classi progettate e implementate devono consentire, oltre all'immissione di una nuova pratica e alla presa in carico di una pratica in coda, la ricerca di una pratica a partire dal codice numerico, la ricerca di tutte le pratiche di un determinato tipo e l'eliminazione di una pratica dato il codice numerico. Il codice Java dovrà generare le eventuali eccezioni ritenute necessarie. Prevedere nella soluzione la possibilità di rendere persistente la coda delle pratiche salvandole in modo ordinato in un file di testo e ricostruendo la coda al momento dell'inizializzazione degli oggetti e realizzare un *main* che consenta all'utente di eseguire le operazioni in modo interattivo.

44 A SCUOLA DI LAVORO Un corriere internazionale che serve esclusivamente clienti abbonati è organizzato in modo che, al momento in cui viene ricevuta una spedizione, identificata dal codice di spedizione assegnato al momento in cui è stata consegnata, vengano visualizzate sul computer dell'incaricato le informazioni (denominazione, indirizzo, città, nazione, telefono, ...) del mittente e del destinatario della spedizione.
Implementare in linguaggio Java le classi *Clienti* e *Spedizioni* che memorizzano, utilizzando ciascuna una tabella a indirizzamento *hash*, rispettivamente oggetti di tipo *Cliente* e *Spedizione* codificando in particolare i metodi che consentono di aggiungere elementi e di ricercare uno specifico elemento generando le eventuali eccezioni necessarie. Implementare inoltre una classe *Corriere* il cui *main* consenta all'utente di eseguire le operazioni in modo interattivo.

45 Nei sistemi CAD e GIS una *polyline* è una lista ordinata di punti A_1, A_2, A_3, ... A_n nel piano cartesiano che approssima una curva, come illustrato negli esempi seguenti:



Progettare mediante un diagramma delle classi UML e implementare in linguaggio Java una classe *Polyline* che consenta di eseguire le seguenti operazioni gestendo in modo adeguato le relative eccezioni:

- aggiunta di un punto in una posizione specifica della *polyline*;
- eliminazione di un punto in una posizione specifica della *polyline*;
- calcolo della lunghezza complessiva della *polyline*;
- determinazione dei vertici del rettangolo che include l'intera *polyline*;
- creazione della *polyline* a partire da un vettore di punti;
- salvataggio e ripristino della *polyline* in un file di tipo testuale;
- concatenazione di due *polyline*.

Realizzare un *main* che consenta all'utente di eseguire le operazioni in modo interattivo.

to chop
frammentare, fare a pezzetti

to depict
rappresentare tramite un disegno

head/tail
testa/coda

parenthesised
messo tra parentesi

sake
letteralmente: motivo, ragione

self-referential
autoreferenziale, che si autoriferisce

to take care of
avere cura di

thread
filamento, filatura

unreachable
non raggiungibile

7. Linked Lists

7.1 Introduction

Linked lists allow us to structure input data sets into elementary units by chopping the data-sets into its many individual elements that are stored in corresponding *cells*. Cells are all chained together into a single thread of cells, starting from the head to the tail. These chained cells can be manipulated dynamically by either *adding* or *removing* elements. These operations can be carried out efficiently (in constant time $O(1)$) by creating new cells or deleting some cells of the list. Linked lists are therefore preferred to arrays whenever we do not know *a priori* the input size. This is all the more interesting for sorting and searching operations that consider dynamic data sets in practice.

7.2 Cells and lists

7.2.1 Illustrating the concepts of cells and lists

Consider an arbitrary set of objects $O = \{O_1, ..., O_n\}$ for which we would like to create a linked list data-structure. A cell is an elementary structure consisting of *two fields*:

- The first field is used for storing the considered object. Thus the first field plays the role of container.
- The second field is used for storing the *reference* to the next cell. This allows it to point to the next cell.

A *linked list* is a set of chained cells that has two distinguished cells: the *head* and the *tail*. The head marks the beginning of the linked list. The tail does not point to the next cell; it thus signals that it is the last cell of the chained cells.

That is, it stores the **null** reference signalling that the cell is a tail.

For example, consider the following linked list of 3 natural numbers $12 \rightarrow 99 \rightarrow 37$. We depict the associated chained cells as follows:

A cell can be interpreted as a pair of (container, reference)

Thus the above linked list can be parenthesised as (12 (99 (37 **null**))).

7.2.2 List as an abstract data-structure

The concept of lists is independent of the programming language. Lists belong to the fundamental ways of structuring data. They can be defined formally in a generic setting as follows:

Constant : Empty list listEmpty (null in Java)

Operations:

Constructor:	List x Object List x Object → List
Head:	List → Object (not defined for listEmpty)
Tail:	List → List (not defined for listEmpty)
isEmpty:	List → Boolean
length:	List → Integer
belongTo:	List x Object → Boolean

7.2.3 Programming linked lists in Java

In Java, we need to create and declare a *new type* induced by a class for creating and initialising elementary cells. Let us denote this class by List. For the sake of simplicity, consider a linked list of integers. The cell container should therefore be of type **int**, and the next field shall refer to the next cell. That cell is also of type List. It is enough to define a variable of type List for pointing to the following cell. More precisely, that

variable stores a reference in Java to that cell (a machine word coded using 32 bits, 4 bytes). Thus we declare the linked list structure by defining the following class:

Program 7.1 Declaration of a linked list

```
class List
{ int container ;
  List next ; }
```

We now need to provide:

- A *proper constructor* to initialise various objects of type List, and
- A few functions implementing the basic operations defined by the abstract framework of §7.2.2.

Program 7.2 Linked list class with constructor and basic functions

```
public class List
{int container ;
List next ;
// Constructor List(head, tail)
List ( int element , List tail )
  { this.container=element ;
    this.next=tail; }
static boolean isEmpty ( List list )
  { if ( list==null ) return true ;
    else return false ; }
static int head ( List list )
  {return list.container ; }
static List tail ( List list )
  {return list.next; }
}
```

In Java, we do not need to explicitly free memory of cells not pointed by any other variables (meaning unreachable) as the garbage collector takes care of that. The type of the class List is recursive since it has one field next that defines a reference to the same type. That is, *a recursive type* is a data type with fields that may contain other values of the same type. The recursive type is self-referential.

[F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Springer-Verlag London Limited, 2009]

QUESTIONS

a What is a recursive type in Java?
b What is a linked list?
c What does the null reference in a linked list mean?
d What are the head and tail of a linked list?

variable stores a reference in Java to that cell (a machine word coded using 32 bits, 4 bytes). Thus we declare the linked list structure by defining the following class:

Program 7.1 Declaration of a linked list

```
class List
{ int container ;
  List next ; }
```

We now need to provide:

- A *proper constructor* to initialise various objects of type List, and
- A few functions implementing the basic operations defined by the abstract framework of §7.2.2.

Program 7.2 Linked list class with constructor and basic functions

```
public class List
{int container ;
List next ;
// Constructor List(head, tail)
List ( int element , List tail )
  { this.container=element ;
    this.next=tail; }
static boolean isEmpty ( List list )
  { if ( list==null ) return true ;
    else return false ; }
static int head ( List list )
  {return list.container ; }
static List tail ( List list )
  {return list.next; }
}
```

In Java, we do not need to explicitly free memory of cells not pointed by any other variables (meaning unreachable) as the garbage collector takes care of that. The type of the class List is recursive since it has one field next that defines a reference to the same type. That is, *a recursive type* is a data type with fields that may contain other values of the same type. The recursive type is self-referential.

[F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Springer-Verlag London Limited, 2009]

QUESTIONS

a What is a recursive type in Java?
b What is a linked list?
c What does the null reference in a linked list mean?
d What are the head and tail of a linked list?

A5

Ereditarietà e polimorfismo

Prendiamo in considerazione la classe *Punto* introdotta nel capitolo A2 e il suo diagramma UML (FIGURA 1).

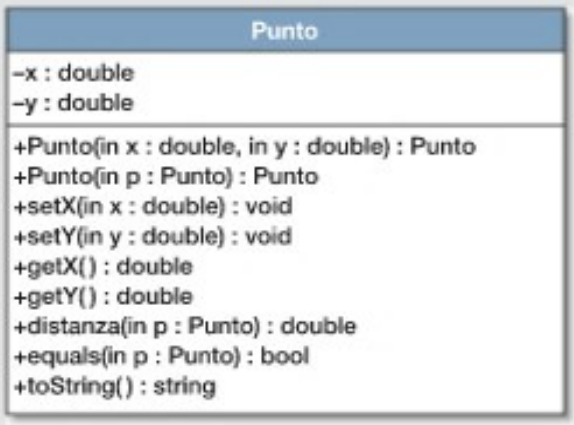


FIGURA 1

Supponiamo adesso di voler definire la classe *PuntoOrientato*, che rispetto alla precedente presenti le seguenti componenti aggiuntive:

- un attributo in più, ovvero la direzione verso cui è orientato il punto (Alto, Basso, Destra, Sinistra con riferimento all'usuale rappresentazione grafica del piano cartesiano);
- un costruttore privo di parametri, uno con parametri per tutti gli attributi e un costruttore di copia;
- due metodi che ruotano il punto rispettivamente in senso orario (verso destra) o antiorario (verso sinistra): quando invocati modificano l'orientamento verso il punto cardinale adiacente a quello corrente;
- un metodo che sposta il punto di una certa distanza in direzione dell'orientamento corrente;

e inoltre riformuli i seguenti metodi per tener conto dell'attributo aggiuntivo:

- *equals* per stabilire se un punto orientato è uguale a un altro;
- *toString* per formattare in una stringa il contenuto informativo di un oggetto di classe *PuntoOrientato*.

In questi casi, piuttosto che definire una nuova classe, è possibile sfruttare la classe esistente (*Punto*) e definire la nuova classe (*PuntoOrientato*) come «estensione» della classe originale. In questo modo *PuntoOrientato* eredita tutti i membri di *Punto* e vi sarà solo la necessità di introdurre

A5

Ereditarietà e polimorfismo

Prendiamo in considerazione la classe *Punto* introdotta nel capitolo A2 e il suo diagramma UML (FIGURA 1).

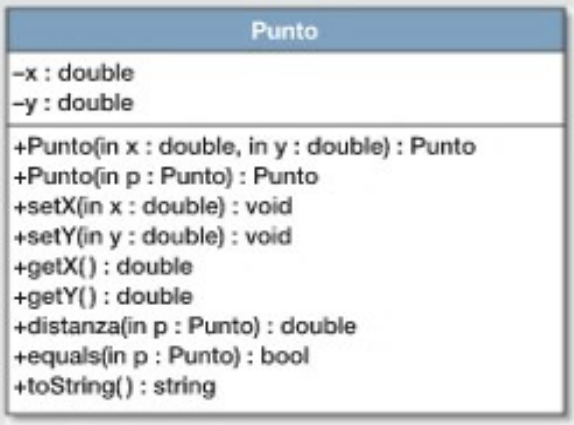


FIGURA 1

Supponiamo adesso di voler definire la classe *PuntoOrientato*, che rispetto alla precedente presenti le seguenti componenti aggiuntive:

- un attributo in più, ovvero la direzione verso cui è orientato il punto (Alto, Basso, Destra, Sinistra con riferimento all'usuale rappresentazione grafica del piano cartesiano);
- un costruttore privo di parametri, uno con parametri per tutti gli attributi e un costruttore di copia;
- due metodi che ruotano il punto rispettivamente in senso orario (verso destra) o antiorario (verso sinistra): quando invocati modificano l'orientamento verso il punto cardinale adiacente a quello corrente;
- un metodo che sposta il punto di una certa distanza in direzione dell'orientamento corrente;

e inoltre riformuli i seguenti metodi per tener conto dell'attributo aggiuntivo:

- *equals* per stabilire se un punto orientato è uguale a un altro;
- *toString* per formattare in una stringa il contenuto informativo di un oggetto di classe *PuntoOrientato*.

In questi casi, piuttosto che definire una nuova classe, è possibile sfruttare la classe esistente (*Punto*) e definire la nuova classe (*PuntoOrientato*) come «estensione» della classe originale. In questo modo *PuntoOrientato* eredita tutti i membri di *Punto* e vi sarà solo la necessità di introdurre

quelli aggiuntivi. La rappresentazione UML di questa situazione è quella di FIGURA 2, dove la generalizzazione, come avevamo visto nel capitolo A1, è rappresentata dalla freccia orientata che unisce la classe derivata alla classe originale.

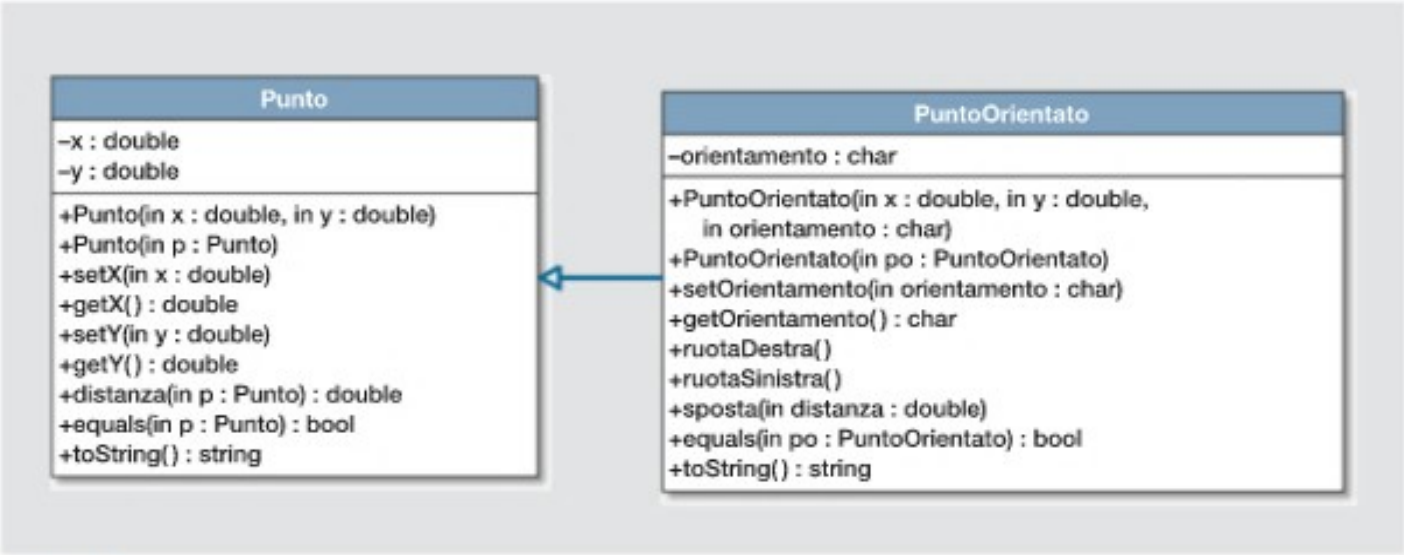


FIGURA 2

Nel linguaggio di programmazione Java la derivazione di una classe a partire da una classe preesistente si ottiene con l'uso della parola chiave **extends**.

ESEMPIO

Senza ridefinire la classe *Punto* introdotta nel capitolo A2, la classe *PuntoOrientato* in Java potrebbe avere la seguente implementazione:

```
public class EccezionePuntoOrientato extends Exception {}

public class PuntoOrientato extends Punto {
  private char orientamento; // nuovo attributo

  // costruttori
  public PuntoOrientato() {
    super(0., 0.);
    this.orientamento = 'A';
  }

  public PuntoOrientato(double x, double y, char orientamento)
        throws EccezionePuntoOrientato {
    super(x, y);
    setOrientamento(orientamento);
  }

  public PuntoOrientato(PuntoOrientato po)
        throws EccezionePuntoOrientato {
    super(po.getX(), po.getY());
    setOrientamento(po.getOrientamento());
  }
  // set/get del nuovo attributo
  public void setOrientamento(char orientamento)
            throws EccezionePuntoOrientato {
    if ((orientamento!='A') && (orientamento!='B') &&
        (orientamento!='D') && (orientamento!='S'))
```

quelli aggiuntivi. La rappresentazione UML di questa situazione è quella di FIGURA 2, dove la generalizzazione, come avevamo visto nel capitolo A1, è rappresentata dalla freccia orientata che unisce la classe derivata alla classe originale.

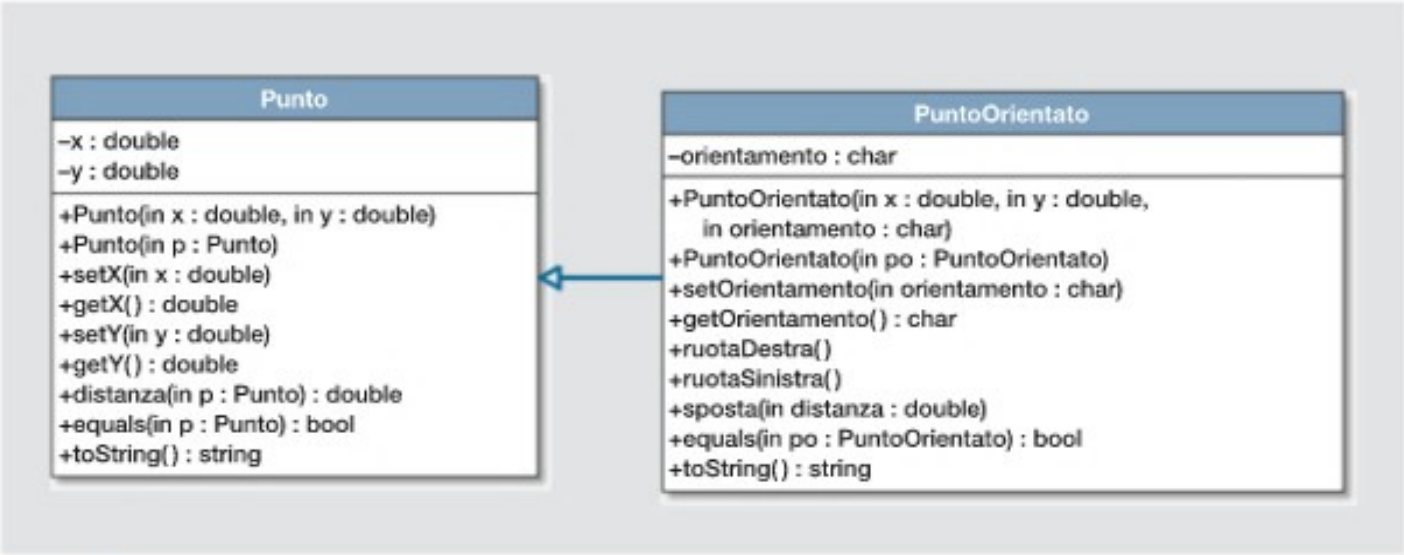


FIGURA 2

Nel linguaggio di programmazione Java la derivazione di una classe a partire da una classe preesistente si ottiene con l'uso della parola chiave **extends**.

ESEMPIO

Senza ridefinire la classe *Punto* introdotta nel capitolo A2, la classe *PuntoOrientato* in Java potrebbe avere la seguente implementazione:

```
public class EccezionePuntoOrientato extends Exception {}

public class PuntoOrientato extends Punto {
  private char orientamento; // nuovo attributo

  // costruttori
  public PuntoOrientato() {
    super(0., 0.);
    this.orientamento = 'A';
  }

  public PuntoOrientato(double x, double y, char orientamento)
        throws EccezionePuntoOrientato {
    super(x, y);
    setOrientamento(orientamento);
  }

  public PuntoOrientato(PuntoOrientato po)
        throws EccezionePuntoOrientato {
    super(po.getX(), po.getY());
    setOrientamento(po.getOrientamento());
  }
  // set/get del nuovo attributo
  public void setOrientamento(char orientamento)
            throws EccezionePuntoOrientato {
    if ((orientamento!='A') && (orientamento!='B') &&
        (orientamento!='D') && (orientamento!='S'))
```

```
        throw new EccezionePuntoOrientato();
      this.orientamento = orientamento;
    }

    public char getOrientamento() {
      return orientamento;
    }

    // altri metodi
    public void ruotaDestra() {
      switch (orientamento) {
            case 'A': orientamento = 'D'; break;
            case 'D': orientamento = 'B'; break;
            case 'B': orientamento = 'S'; break;
            case 'S': orientamento = 'A'; break;
      }
    }

    public void ruotaSinistra() {
      switch (orientamento) {
            case 'A': orientamento = 'S'; break;
            case 'D': orientamento = 'A'; break;
            case 'B': orientamento = 'D'; break;
            case 'S': orientamento = 'B'; break;
      }
    }

    public void sposta(double distanza){
      switch (orientamento) {
            case 'A': setY(getY() + distanza); break;
            case 'B': setY(getY() - distanza); break;
            case 'S': setX(getX() - distanza); break;
            case 'D': setX(getX() + distanza); break;
      }
    }

    public boolean equals(PuntoOrientato po) {
      return (super.getX() == po.getX() &&
              super.getY() == po.getY() &&
              this.getOrientamento() == po.getOrientamento());
    }

    public String toString() {
      return super.toString()+orientamento;
    }
  }
```

Il seguente metodo *main* consente di testare le funzionalità della classe *PuntoOrientato*

```
  public static void main(String[] args) {
    PuntoOrientato po1 = new PuntoOrientato();
    PuntoOrientato po2 = new PuntoOrientato();
    PuntoOrientato po3 = new PuntoOrientato();
    PuntoOrientato po4 = new PuntoOrientato();

    try {
      po1 = new PuntoOrientato(1.,1.,'D');
      po2 = new PuntoOrientato(2.,2.,'A');
      po3 = new PuntoOrientato(po1);
```

```
        throw new EccezionePuntoOrientato();
      this.orientamento = orientamento;
    }

    public char getOrientamento() {
      return orientamento;
    }

    // altri metodi
    public void ruotaDestra() {
      switch (orientamento) {
            case 'A': orientamento = 'D'; break;
            case 'D': orientamento = 'B'; break;
            case 'B': orientamento = 'S'; break;
            case 'S': orientamento = 'A'; break;
      }
    }

    public void ruotaSinistra() {
      switch (orientamento) {
            case 'A': orientamento = 'S'; break;
            case 'D': orientamento = 'A'; break;
            case 'B': orientamento = 'D'; break;
            case 'S': orientamento = 'B'; break;
      }
    }

    public void sposta(double distanza){
      switch (orientamento) {
            case 'A': setY(getY() + distanza); break;
            case 'B': setY(getY() - distanza); break;
            case 'S': setX(getX() - distanza); break;
            case 'D': setX(getX() + distanza); break;
      }
    }

    public boolean equals(PuntoOrientato po) {
      return (super.getX() == po.getX() &&
              super.getY() == po.getY() &&
              this.getOrientamento() == po.getOrientamento());
    }

    public String toString() {
      return super.toString()+orientamento;
    }
  }
```

Il seguente metodo *main* consente di testare le funzionalità della classe *PuntoOrientato*

```
  public static void main(String[] args) {
    PuntoOrientato po1 = new PuntoOrientato();
    PuntoOrientato po2 = new PuntoOrientato();
    PuntoOrientato po3 = new PuntoOrientato();
    PuntoOrientato po4 = new PuntoOrientato();

    try {
      po1 = new PuntoOrientato(1.,1.,'D');
      po2 = new PuntoOrientato(2.,2.,'A');
      po3 = new PuntoOrientato(po1);
```

```
      po4 = new PuntoOrientato(0.,0.,'X');
    }
    catch (EccezionePuntoOrientato eccezione) {
      System.out.println("Generata eccezione");
    }

    System.out.println("PO1 = "+po1.toString());
    System.out.println("PO2 = "+po2.toString());
    System.out.println("PO3 = "+po3.toString());
    System.out.println("Distanza PO1-PO2: "+po1.distanza(po2));
    System.out.println("Distanza PO1-PO3: "+po1.distanza(po3));

    if (po1.equals(po3))
      System.out.println("PO1 e PO3 coincidono");
    else
      System.out.println("PO1 e PO3 NON coincidono");
    po1.ruotaSinistra();
    System.out.println("PO1 = "+po1.toString());
    po1.sposta(10.0);
    System.out.println("PO1 = "+po1.toString());
  }
```

e produce il seguente output:

```
Generata eccezione
PO1 = (1.0;1.0)D
PO2 = (2.0;2.0)A
PO3 = (1.0;1.0)D
Distanza PO1-PO2: 1.4142135623730951
Distanza PO1-PO3: 0.0
PO1 e PO3 coincidono
PO1 = (1.0;1.0)A
PO1 = (1.0;11.0)A
```

La parola chiave **super** nell'esempio precedente riferisce gli attributi e i metodi – compreso il costruttore – della classe *Punto* da cui la classe *PuntoOrientato* deriva.

1 Classi derivate; *overriding* e *overloading* dei metodi

L'**ereditarietà** è un meccanismo di astrazione finalizzato alla creazione di gerarchie di classi: con essa si ha la possibilità di riutilizzare codice comune a più classi della stessa gerarchia.
Nell'esempio di apertura abbiamo visto come, tramite la parola chiave **extends**, sia possibile dichiarare una nuova classe come estensione di una classe già esistente da cui la nuova classe eredita i membri (attributi e metodi). La classe che viene estesa è denominata **superclasse** e rappresenta una generalizzazione della **sottoclasse** derivata, che è invece una

```
      po4 = new PuntoOrientato(0.,0.,'X');
    }
    catch (EccezionePuntoOrientato eccezione) {
      System.out.println("Generata eccezione");
    }

    System.out.println("PO1 = "+po1.toString());
    System.out.println("PO2 = "+po2.toString());
    System.out.println("PO3 = "+po3.toString());
    System.out.println("Distanza PO1-PO2: "+po1.distanza(po2));
    System.out.println("Distanza PO1-PO3: "+po1.distanza(po3));

    if (po1.equals(po3))
      System.out.println("PO1 e PO3 coincidono");
    else
      System.out.println("PO1 e PO3 NON coincidono");
    po1.ruotaSinistra();
    System.out.println("PO1 = "+po1.toString());
    po1.sposta(10.0);
    System.out.println("PO1 = "+po1.toString());
  }
```

e produce il seguente output:

```
Generata eccezione
PO1 = (1.0;1.0)D
PO2 = (2.0;2.0)A
PO3 = (1.0;1.0)D
Distanza PO1-PO2: 1.4142135623730951
Distanza PO1-PO3: 0.0
PO1 e PO3 coincidono
PO1 = (1.0;1.0)A
PO1 = (1.0;11.0)A
```

La parola chiave **super** nell'esempio precedente riferisce gli attributi e i metodi – compreso il costruttore – della classe *Punto* da cui la classe *PuntoOrientato* deriva.

1 Classi derivate; *overriding* e *overloading* dei metodi

L'**ereditarietà** è un meccanismo di astrazione finalizzato alla creazione di gerarchie di classi: con essa si ha la possibilità di riutilizzare codice comune a più classi della stessa gerarchia.
Nell'esempio di apertura abbiamo visto come, tramite la parola chiave **extends**, sia possibile dichiarare una nuova classe come estensione di una classe già esistente da cui la nuova classe eredita i membri (attributi e metodi). La classe che viene estesa è denominata **superclasse** e rappresenta una generalizzazione della **sottoclasse** derivata, che è invece una

specializzazione della classe originale. La classe derivata può aggiungere nuovi membri a quelli ereditati dalla superclasse, o ridefinirne alcuni.
La parola chiave `final` del linguaggio Java applicata a una classe impedisce di estenderla per ereditarietà.

I costruttori di una classe **non** sono ereditati dalle classi derivate. Queste debbono necessariamente ridefinire i propri costruttori.

Analizzando la classe *PuntoOrientato* dell'esempio di apertura si può osservare che:

- definisce il nuovo attributo (nuovo rispetto agli attributi della classe originale *Punto*) *orientamento* di tipo carattere che consente di codificare la direzione di orientamento del punto; gli attributi *x* e *y* non devono essere nuovamente definiti perché sono ereditati dalla classe *Punto*;
- il metodo *setOrientamento* accetta come valori validi i caratteri «A» (Alto), «B» (Basso), «D» (Destra) ed «S» (Sinistra): nel caso sia fornito un carattere diverso viene generata una specifica eccezione;
- i metodi *equals* e *toString* sono ridefiniti rispetto ai metodi omonimi della classe *Punto* per includere la gestione del nuovo attributo *orientamento*.

Sia nella classe *Punto* sia nella classe derivata *PuntoOrientato* vi sono più costruttori: tutti hanno lo stesso nome e, come abbiamo già visto nei capitoli precedenti, sarà il compilatore a selezionare quale utilizzare in funzione degli argomenti specificati al momento dell'invocazione. Questa tecnica è denominata ***overloading*** e consente di definire più metodi (non esclusivamente i costruttori) nell'ambito della stessa classe aventi lo stesso nome, a patto che differiscano nella firma per numero e tipo dei parametri specificati.
La sola modifica del tipo del valore di ritorno non è sufficiente a realizzare un *overloading* e non è accettata dal compilatore.

La tecnica di ridefinizione in una classe derivata di metodi già presenti nella superclasse è denominata ***overriding***: un metodo sovrascritto nasconde – negli oggetti istanza della classe derivata – la definizione data nella superclasse sostituendola con quella data nella classe derivata. La parola chiave `final` del linguaggio Java applicata a un metodo di una classe ne impedisce la ridefinizione in una classe derivata.

Un metodo ridefinito in una classe derivata nasconde un metodo omonimo della superclasse solo se la sua firma è esattamente la stessa del metodo originale; nell'esempio della classe *PuntoOrientato* il metodo *toString* effettua l'*overriding* del metodo omonimo della classe *Punto*, mentre il metodo *equals* originale rimane disponibile anche per gli oggetti di classe *PuntoOrientato* in *overloading* con il metodo originale ereditato.

Il riuso del software

Il meccanismo di derivazione delle classi, e la conseguente ereditarietà di metodi e attributi, è la modalità con cui i linguaggi di programmazione OO consentono di realizzare la pratica del riuso del software.
Il riuso di software esistente – già progettato, implementato e testato – nella realizzazione di nuovo software è considerata una pratica fondamentale per il rispetto dei tempi e dei costi di sviluppo da parte di molte aziende e organizzazioni.
La realizzazione di «librerie» di classi estendibili, e quindi facilmente adattabili a nuovi contesti utilizzando il meccanismo dell'ereditarietà del codice, ha rivoluzionato la pratica dello sviluppo software.

specializzazione della classe originale. La classe derivata può aggiungere nuovi membri a quelli ereditati dalla superclasse, o ridefinirne alcuni. La parola chiave `final` del linguaggio Java applicata a una classe impedisce di estenderla per ereditarietà.

I costruttori di una classe **non** sono ereditati dalle classi derivate. Queste debbono necessariamente ridefinire i propri costruttori.

Analizzando la classe *PuntoOrientato* dell'esempio di apertura si può osservare che:

- definisce il nuovo attributo (nuovo rispetto agli attributi della classe originale *Punto*) *orientamento* di tipo carattere che consente di codificare la direzione di orientamento del punto; gli attributi *x* e *y* non devono essere nuovamente definiti perché sono ereditati dalla classe *Punto*;
- il metodo *setOrientamento* accetta come valori validi i caratteri «A» (Alto), «B» (Basso), «D» (Destra) ed «S» (Sinistra): nel caso sia fornito un carattere diverso viene generata una specifica eccezione;
- i metodi *equals* e *toString* sono ridefiniti rispetto ai metodi omonimi della classe *Punto* per includere la gestione del nuovo attributo *orientamento*.

Sia nella classe *Punto* sia nella classe derivata *PuntoOrientato* vi sono più costruttori: tutti hanno lo stesso nome e, come abbiamo già visto nei capitoli precedenti, sarà il compilatore a selezionare quale utilizzare in funzione degli argomenti specificati al momento dell'invocazione. Questa tecnica è denominata ***overloading*** e consente di definire più metodi (non esclusivamente i costruttori) nell'ambito della stessa classe aventi lo stesso nome, a patto che differiscano nella firma per numero e tipo dei parametri specificati.
La sola modifica del tipo del valore di ritorno non è sufficiente a realizzare un *overloading* e non è accettata dal compilatore.

La tecnica di ridefinizione in una classe derivata di metodi già presenti nella superclasse è denominata ***overriding***: un metodo sovrascritto nasconde – negli oggetti istanza della classe derivata – la definizione data nella superclasse sostituendola con quella data nella classe derivata. La parola chiave `final` del linguaggio Java applicata a un metodo di una classe ne impedisce la ridefinizione in una classe derivata.

Un metodo ridefinito in una classe derivata nasconde un metodo omonimo della superclasse solo se la sua firma è esattamente la stessa del metodo originale; nell'esempio della classe *PuntoOrientato* il metodo *toString* effettua l'*overriding* del metodo omonimo della classe *Punto*, mentre il metodo *equals* originale rimane disponibile anche per gli oggetti di classe *PuntoOrientato* in *overloading* con il metodo originale ereditato.

Il riuso del software

Il meccanismo di derivazione delle classi, e la conseguente ereditarietà di metodi e attributi, è la modalità con cui i linguaggi di programmazione OO consentono di realizzare la pratica del riuso del software.
Il riuso di software esistente – già progettato, implementato e testato – nella realizzazione di nuovo software è considerata una pratica fondamentale per il rispetto dei tempi e dei costi di sviluppo da parte di molte aziende e organizzazioni.
La realizzazione di «librerie» di classi estendibili, e quindi facilmente adattabili a nuovi contesti utilizzando il meccanismo dell'ereditarietà del codice, ha rivoluzionato la pratica dello sviluppo software.

Il linguaggio Java definisce una classe «radice» da cui derivano implicitamente tutte le altre: la classe ***Object***.

Tale classe prevede tra i suoi metodi anche i metodi *toString* ed *equals*. Quando il compilatore Java deve convertire un oggetto in una stringa, invoca il metodo *toString* della classe di cui l'oggetto è istanza. Se il programmatore non ha ridefinito nella propria classe il metodo *toString*, viene invocato il metodo originale definito nella classe *Object* e automaticamente ereditato da qualsiasi classe Java: esso genera una stringa che è spesso di scarsa utilità; per ottenere un risultato che abbia un senso è necessario ridefinire nella propria classe un appropriato metodo *toString* effettuandone l'*overriding*.

ESEMPIO Se nella classe *Punto* non fosse stato definito un metodo *toString*, il seguente frammento di codice

```
...
Punto p = new Punto(0.,0.);
System.out.println(p);
...
```

produrrebbe un output del tipo

```
Punto@3e25a5
```

invece di

```
(0.0;0.0)
```

che è la stringa formattata dal metodo ridefinito. Infatti il risultato prodotto dal metodo *toString* della classe *Object* è una stringa che, oltre al nome della classe di cui l'oggetto è istanza, comprende la codifica esadecimale del codice *hash* dell'oggetto stesso, così come fornita dal metodo *hashCode* della stessa classe *Object*.

Il metodo *hashCode* della classe *Object* restituisce la codifica *hash* dell'indirizzo di memorizzazione nello *heap* dell'oggetto su cui è invocato. Dato che anche il metodo *equals* definito nella classe *Object* opera sui risultati del metodo *hashCode*, esso considera uguali due oggetti solo se sono lo stesso oggetto: due oggetti distinti, infatti, anche se con tutti gli attributi tra loro identici, hanno necessariamente indirizzi diversi e il risultato del loro confronto sarà sempre falso!

In generale l'ereditarietà viene utilizzata per fattorizzare (raggruppare) attributi comuni a più classi, evitando inutili ridondanze.

ESEMPIO Per sviluppare un'applicazione per la gestione del personale della scuola è necessario definire le classi *Impiegato* e *Docente*. Tra di esse è possibile individuare attributi comuni a tutti i dipendenti, come il nominativo, il sesso e l'indirizzo, e attributi specifici, come l'ufficio per gli impiegati o come il ruolo (supplente, insegnante diplomato, insegnante laureato, ...) e la disciplina di insegnamento per i docenti. Per evitare inutili ridondanze si può introdurre una classe *Dipendente* per la gestione degli attributi comuni: le classi *Impiegato* e *Docente* ▶

Il linguaggio Java definisce una classe «radice» da cui derivano implicitamente tutte le altre: la classe ***Object***.

Tale classe prevede tra i suoi metodi anche i metodi *toString* ed *equals*. Quando il compilatore Java deve convertire un oggetto in una stringa, invoca il metodo *toString* della classe di cui l'oggetto è istanza. Se il programmatore non ha ridefinito nella propria classe il metodo *toString*, viene invocato il metodo originale definito nella classe *Object* e automaticamente ereditato da qualsiasi classe Java: esso genera una stringa che è spesso di scarsa utilità; per ottenere un risultato che abbia un senso è necessario ridefinire nella propria classe un appropriato metodo *toString* effettuandone l'*overriding*.

ESEMPIO Se nella classe *Punto* non fosse stato definito un metodo *toString*, il seguente frammento di codice

```
...
Punto p = new Punto(0.,0.);
System.out.println(p);
...
```

produrrebbe un output del tipo

```
Punto@3e25a5
```

invece di

```
(0.0;0.0)
```

che è la stringa formattata dal metodo ridefinito. Infatti il risultato prodotto dal metodo *toString* della classe *Object* è una stringa che, oltre al nome della classe di cui l'oggetto è istanza, comprende la codifica esadecimale del codice *hash* dell'oggetto stesso, così come fornita dal metodo *hashCode* della stessa classe *Object*.

Il metodo *hashCode* della classe *Object* restituisce la codifica *hash* dell'indirizzo di memorizzazione nello *heap* dell'oggetto su cui è invocato. Dato che anche il metodo *equals* definito nella classe *Object* opera sui risultati del metodo *hashCode*, esso considera uguali due oggetti solo se sono lo stesso oggetto: due oggetti distinti, infatti, anche se con tutti gli attributi tra loro identici, hanno necessariamente indirizzi diversi e il risultato del loro confronto sarà sempre falso!

In generale l'ereditarietà viene utilizzata per fattorizzare (raggruppare) attributi comuni a più classi, evitando inutili ridondanze.

ESEMPIO Per sviluppare un'applicazione per la gestione del personale della scuola è necessario definire le classi *Impiegato* e *Docente*. Tra di esse è possibile individuare attributi comuni a tutti i dipendenti, come il nominativo, il sesso e l'indirizzo, e attributi specifici, come l'ufficio per gli impiegati o come il ruolo (supplente, insegnante diplomato, insegnante laureato, ...) e la disciplina di insegnamento per i docenti. Per evitare inutili ridondanze si può introdurre una classe *Dipendente* per la gestione degli attributi comuni: le classi *Impiegato* e *Docente* ▶

sono definite come estensioni di *Dipendente* e comprendono gli attributi specifici. Il diagramma UML delle classi di FIGURA 3 rappresenta questa soluzione.

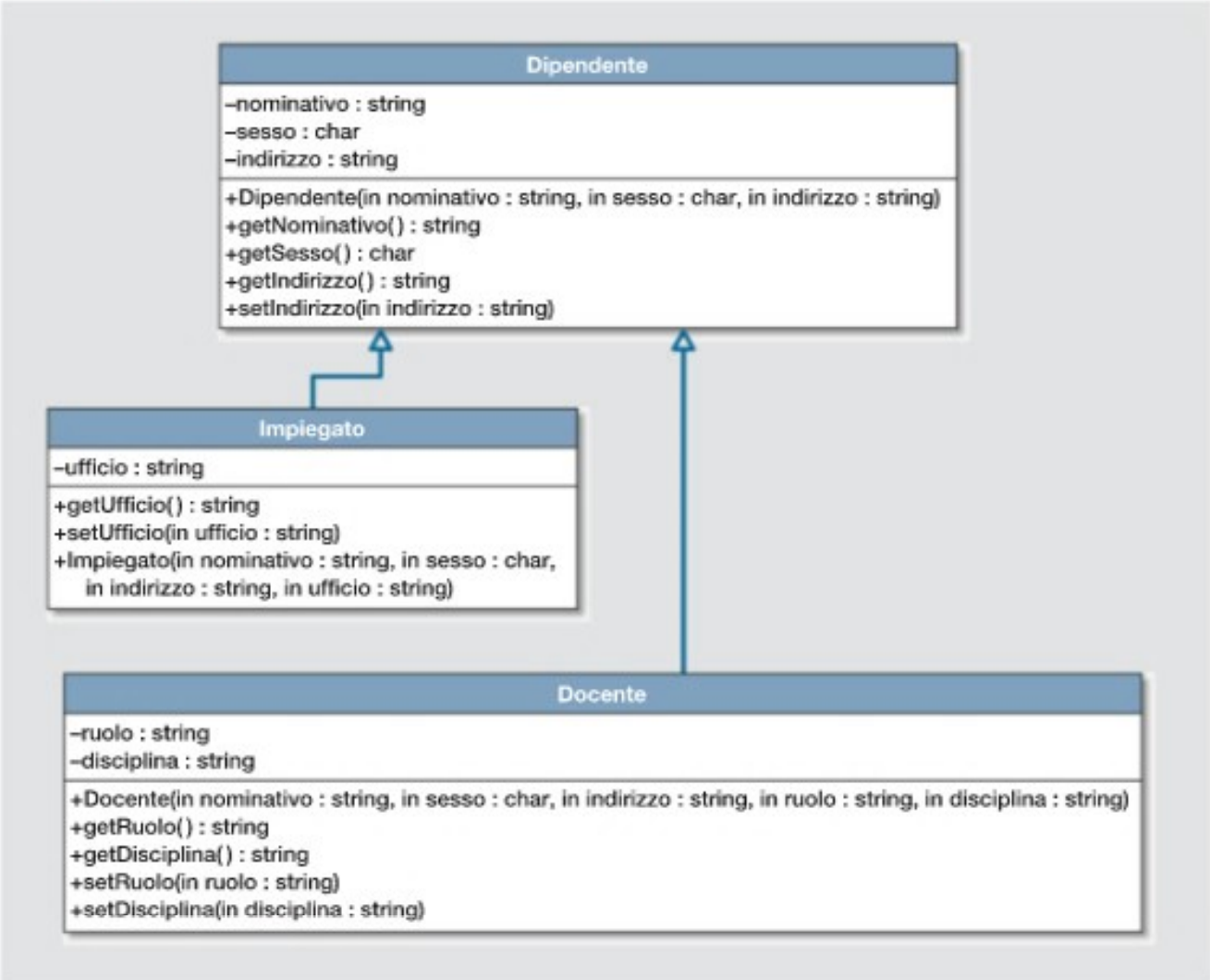


FIGURA 3

In linguaggio Java tale soluzione può essere così implementata:

```
public class Dipendente {
  private String nominativo;
  private char sesso;
  private String indirizzo;

  public Dipendente(String nominativo, char sesso, String indirizzo) {
    this.nominativo = nominativo;
    this.sesso = sesso;
    this.indirizzo = indirizzo;
  }

  public String getIndirizzo() {
    return indirizzo;
  }

  public void setIndirizzo(String indirizzo) {
    this.indirizzo = indirizzo;
  }

  public String getNominativo() {
    return nominativo;
  }
```

sono definite come estensioni di *Dipendente* e comprendono gli attributi specifici. Il diagramma UML delle classi di FIGURA 3 rappresenta questa soluzione.

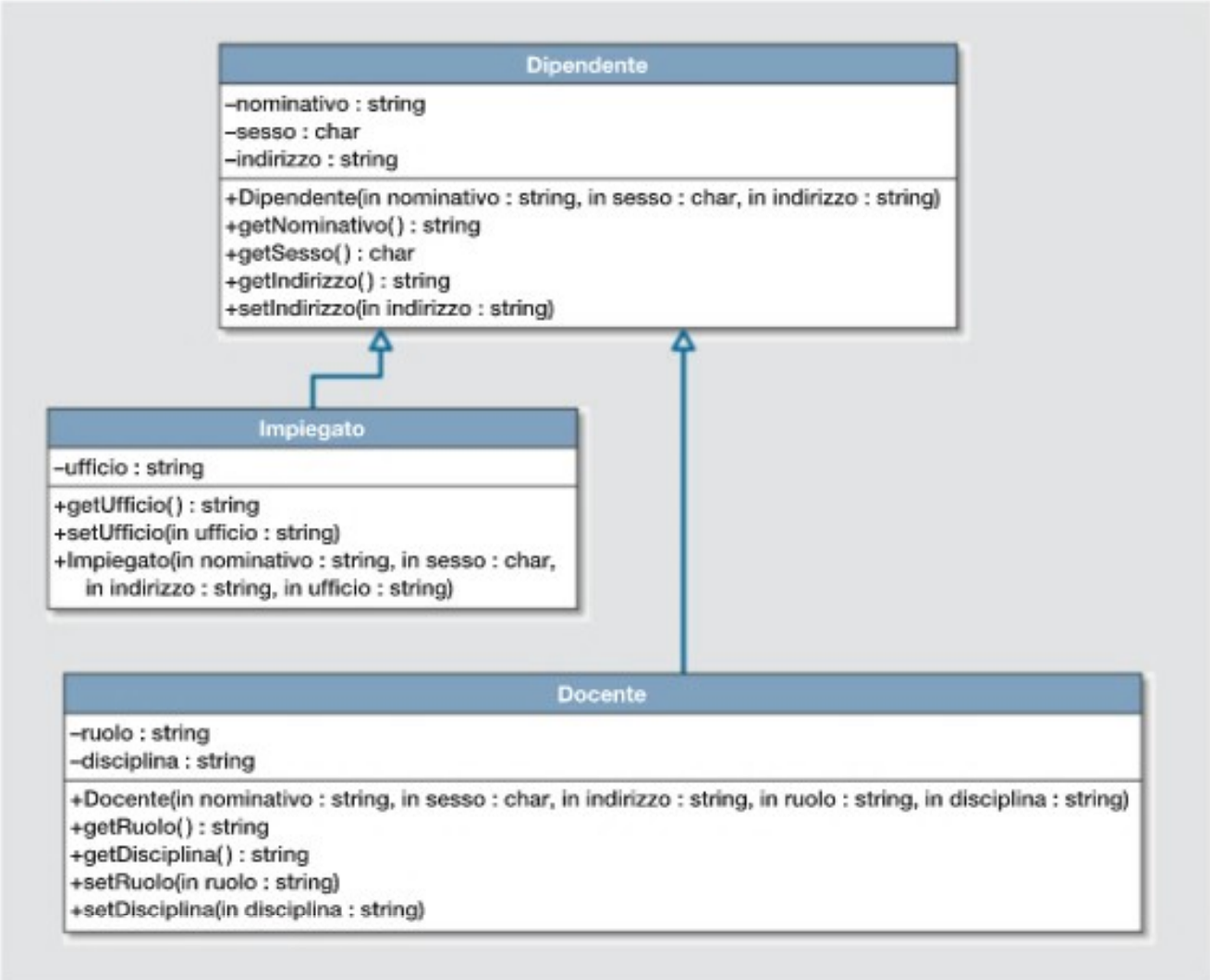


FIGURA 3

In linguaggio Java tale soluzione può essere così implementata:

```
public class Dipendente {
  private String nominativo;
  private char sesso;
  private String indirizzo;

  public Dipendente(String nominativo, char sesso, String indirizzo) {
    this.nominativo = nominativo;
    this.sesso = sesso;
    this.indirizzo = indirizzo;
  }

  public String getIndirizzo() {
    return indirizzo;
  }

  public void setIndirizzo(String indirizzo) {
    this.indirizzo = indirizzo;
  }

  public String getNominativo() {
    return nominativo;
  }
```

```
    public char getSesso() {
      return sesso;
    }
  }

  public class Impiegato extends Dipendente {
    private String ufficio;

    public Impiegato(String nominativo, char sesso, String indirizzo,
                     String ufficio) {
      super(nominativo, sesso, indirizzo);
      this.ufficio = ufficio;
    }

    public String getUfficio() {
      return ufficio;
    }

    public void setUfficio(String ufficio) {
      this.ufficio = ufficio;
    }
  }

  public class Docente extends Dipendente {
    private String ruolo;
    private String disciplina;

    public Docente(String nominativo, char sesso, String indirizzo,
                   String ruolo, String disciplina) {
      super(nominativo, sesso, indirizzo);
      this.ruolo = ruolo;
      this.disciplina = disciplina;
    }

    public String getDisciplina() {
      return disciplina;
    }

    public void setDisciplina(String disciplina) {
      this.disciplina = disciplina;
    }

    public String getRuolo() {
      return ruolo;
    }

    public void setRuolo(String ruolo) {
      this.ruolo = ruolo;
    }
  }
```

Il costruttore delle classi derivate (*Impiegato*, *Docente*) deve inizializzare tutti gli attributi di un oggetto istanza della classe: ha quindi un parametro per ogni attributo ereditato dalla superclasse (*Dipendente*) più un parametro per ogni nuovo attributo definito nella sottoclasse.
Dato che la parola chiave **super** nel linguaggio Java riferisce la superclasse e che i costruttori hanno per definizione il nome della classe in cui

```
    public char getSesso() {
      return sesso;
    }
  }

  public class Impiegato extends Dipendente {
    private String ufficio;

    public Impiegato(String nominativo, char sesso, String indirizzo,
                     String ufficio) {
      super(nominativo, sesso, indirizzo);
      this.ufficio = ufficio;
    }

    public String getUfficio() {
      return ufficio;
    }

    public void setUfficio(String ufficio) {
      this.ufficio = ufficio;
    }
  }

  public class Docente extends Dipendente {
    private String ruolo;
    private String disciplina;

    public Docente(String nominativo, char sesso, String indirizzo,
                   String ruolo, String disciplina) {
      super(nominativo, sesso, indirizzo);
      this.ruolo = ruolo;
      this.disciplina = disciplina;
    }

    public String getDisciplina() {
      return disciplina;
    }

    public void setDisciplina(String disciplina) {
      this.disciplina = disciplina;
    }

    public String getRuolo() {
      return ruolo;
    }

    public void setRuolo(String ruolo) {
      this.ruolo = ruolo;
    }
  }
```

Il costruttore delle classi derivate (*Impiegato*, *Docente*) deve inizializzare tutti gli attributi di un oggetto istanza della classe: ha quindi un parametro per ogni attributo ereditato dalla superclasse (*Dipendente*) più un parametro per ogni nuovo attributo definito nella sottoclasse.
Dato che la parola chiave **super** nel linguaggio Java riferisce la superclasse e che i costruttori hanno per definizione il nome della classe in cui

sono definiti, il costruttore della superclasse viene normalmente invocato nel costruttore della sottoclasse mediante la seguente sintassi:

```
super(...);
```

Questa invocazione, se presente, deve essere la prima istruzione del costruttore di una classe derivata.

Riprendendo l'analisi del codice della classe *PuntoOrientato* si può osservare quanto segue.

- I primi due costruttori hanno come prima istruzione l'invocazione del costruttore della superclasse `super(...,...);` a cui sono forniti come parametri i valori delle coordinate che il codice del costruttore della classe *Punto* assegna agli attributi *x* e *y*. Il codice dei costruttori provvede di seguito ad assegnare il valore all'attributo *orientamento* invocando il metodo *setOrientamento*.
- Il terzo costruttore è un costruttore di copia e anch'esso ricorre all'uso della parola chiave `super` nella prima istruzione per invocare il costruttore della superclasse, ma in questo caso – dovendo copiare i valori degli attributi privati della classe *Punto* ereditati dalla classe *PuntoOrientato* dell'oggetto *po* passato come argomento – utilizza i metodi pubblici *getX* e *getY* per accedervi.

Anche se ereditati, gli attributi di livello `private` non sono mai accessibili dal codice dei metodi di una classe diversa da quella in cui sono dichiarati. Come è stato illustrato nel capitolo A2, per risultare accessibili dal codice dei metodi di una classe derivata, gli attributi devono essere dichiarati di livello `protected`.

- Nel metodo ridefinito *toString* l'espressione `super.toString()` riferisce il risultato prodotto dall'omonimo metodo della classe *Punto* per ottenere una stringa formattata con i valori numerici delle due coordinate cartesiane (per esempio `(1.0;1.0)`) a cui concatenare il valore dell'attributo *orientamento* che indica la direzione verso cui un oggetto di tipo *PuntoOrientato* è rivolto (per esempio `(1.0;1.0)A`).
- Nel metodo *equals* le espressioni `super.getX()` e `super.getY()` riferiscono i valori restituiti dagli omonimi metodi della classe *Punto* per essere confrontati con i valori restituiti dai corrispondenti metodi ereditati dalla classe *PuntoOrientato* applicati all'oggetto *po* fornito come parametro.

In realtà i metodi *getX()* e *getY()* ereditati dalla classe *PuntoOrientato* sono di livello `public` e, non essendo ridefiniti, possono essere invocati dal codice dei metodi di una classe qualsiasi. In questo caso non è necessario premettere la parola chiave `super` (potrebbe infatti essere indifferentemente usata la parola chiave `this`).

sono definiti, il costruttore della superclasse viene normalmente invocato nel costruttore della sottoclasse mediante la seguente sintassi:

```
super(...);
```

Questa invocazione, se presente, deve essere la prima istruzione del costruttore di una classe derivata.

Riprendendo l'analisi del codice della classe *PuntoOrientato* si può osservare quanto segue.

- I primi due costruttori hanno come prima istruzione l'invocazione del costruttore della superclasse `super(...,...);` a cui sono forniti come parametri i valori delle coordinate che il codice del costruttore della classe *Punto* assegna agli attributi *x* e *y*. Il codice dei costruttori provvede di seguito ad assegnare il valore all'attributo *orientamento* invocando il metodo *setOrientamento*.
- Il terzo costruttore è un costruttore di copia e anch'esso ricorre all'uso della parola chiave `super` nella prima istruzione per invocare il costruttore della superclasse, ma in questo caso – dovendo copiare i valori degli attributi privati della classe *Punto* ereditati dalla classe *PuntoOrientato* dell'oggetto *po* passato come argomento – utilizza i metodi pubblici *getX* e *getY* per accedervi.

Anche se ereditati, gli attributi di livello `private` non sono mai accessibili dal codice dei metodi di una classe diversa da quella in cui sono dichiarati. Come è stato illustrato nel capitolo A2, per risultare accessibili dal codice dei metodi di una classe derivata, gli attributi devono essere dichiarati di livello `protected`.

- Nel metodo ridefinito *toString* l'espressione `super.toString()` riferisce il risultato prodotto dall'omonimo metodo della classe *Punto* per ottenere una stringa formattata con i valori numerici delle due coordinate cartesiane (per esempio `(1.0;1.0)`) a cui concatenare il valore dell'attributo *orientamento* che indica la direzione verso cui un oggetto di tipo *PuntoOrientato* è rivolto (per esempio `(1.0;1.0)A`).
- Nel metodo *equals* le espressioni `super.getX()` e `super.getY()` riferiscono i valori restituiti dagli omonimi metodi della classe *Punto* per essere confrontati con i valori restituiti dai corrispondenti metodi ereditati dalla classe *PuntoOrientato* applicati all'oggetto *po* fornito come parametro.

In realtà i metodi *getX()* e *getY()* ereditati dalla classe *PuntoOrientato* sono di livello `public` e, non essendo ridefiniti, possono essere invocati dal codice dei metodi di una classe qualsiasi. In questo caso non è necessario premettere la parola chiave `super` (potrebbe infatti essere indifferentemente usata la parola chiave `this`).

L'*overriding* fa in modo che un metodo sovrascritto nasconda nella classe derivata la definizione data nella superclasse per un metodo avente la stessa firma, tuttavia la forma originale risulta accessibile nel codice dei metodi della classe derivata utilizzando la parola chiave **`super`** mediante la quale risultano invocabili anche i costruttori della superclasse.
La possibilità di accedere ai costruttori e ai metodi della classe originale utilizzando la parola chiave **`super`** garantisce il rispetto del principio di incapsulamento rispetto alle classi derivate: eventuali modifiche del codice dei metodi della superclasse che non ne modifichino la firma non si rifletteranno sul codice dei metodi della sottoclasse.

2 Gerarchie di classi: *up-casting* e *down-casting* di oggetti

In una gerarchia di classi derivate è possibile assegnare un oggetto istanza di una classe della gerarchia a un riferimento avente come tipo una qualsiasi superclasse, ma non viceversa.

ESEMPIO Riferendosi alla gerarchia di classi *Dipendente*, *Impiegato* e *Docente* definite nel paragrafo precedente, a un riferimento di tipo *Dipendente* può essere assegnato un oggetto istanza della classe *Impiegato* (o *Docente*), ma a un riferimento di tipo *Docente* non può essere assegnato un oggetto istanza della classe *Dipendente* (o *Impiegato*).

La regola precedente stabilisce semplicemente che l'assegnamento di un oggetto a un riferimento deve garantire la corretta invocazione di tutti i metodi definiti dalla classe del riferimento, cosa che è sempre vera nel caso delle sottoclassi, ma che non lo è nel caso delle superclassi.

ESEMPIO Supponendo di avere un oggetto *po* istanza della classe *PuntoOrientato*, un assegnamento come il seguente è corretto:

```
Punto p = po;
```

Infatti l'oggetto *po* riferito da *p* ha tutti i requisiti per poter rispondere ai metodi della classe *Punto* che potrebbero essere invocati. Viceversa, disponendo di un oggetto *p* istanza della classe *Punto*, l'assegnamento

```
PuntoOrientato po = p;
```

risulta errato, in quanto l'oggetto riferito da *po* non sarebbe in grado di rispondere a metodi quali *setOrientamento*, *getOrientamento*, *ruotaDestra*, *ruotaSinistra* e *sposta*, invocabili su un qualsiasi riferimento di classe *PuntoOrientato*. Il compilatore del linguaggio Java genera in questo caso un errore rilevando l'incompatibilità dei tipi.

Rispettando la regola enunciata in precedenza il linguaggio Java consente di trasformare il tipo degli oggetti mediante operazioni di ***casting***. L'operatore di *casting* viene premesso al riferimento dell'oggetto e ha il seguente formato

```
(tipo)oggetto
```

dove *tipo* è il nuovo tipo di *oggetto*.

Il principio di sostituzione di Liskov e la relazione «*is a*»

La relazione di specializzazione di una classe derivata rispetto alla superclasse da cui deriva è spesso denominata relazione «*is a*», per sottolineare come un'istanza di una classe è anche un'istanza della superclasse.
Nel 1987 Barbara Liskov enunciò il **principio di sostituzione**, che si applica ai linguaggi di programmazione che realizzano il polimorfismo e che è oggi noto con il suo nome.
Esso afferma che un oggetto il cui tipo è un sottotipo di *T* deve poter essere utilizzato in un qualsiasi contesto in cui è legale utilizzare un oggetto di tipo *T*. La definizione di **sottotipo** data da Liskov è piuttosto complessa, ma coincide con quella di classe derivata della maggior parte dei linguaggi di programmazione OO.

ESEMPIO

Il metodo *equals* della classe *PuntoOrientato* potrebbe essere così codificato:

```
public boolean equals(PuntoOrientato po) {
   return (super.equals((Punto)po) &&
   getOrientamento() == po.getOrientamento());
}
```

L'invocazione del metodo *equals* della superclasse *Punto* richiede il passaggio di un argomento di tipo *Punto*, cosa che si ottiene effettuando un *casting* dell'oggetto *po* fornito come parametro. Essendo *Punto* una superclasse di *PuntoOrientato*, questa operazione è lecita: il metodo confronta gli attributi *x* e *y* che la classe *PuntoOrientato* eredita dalla classe *Punto*.

Nel rispetto della regola che impedisce che un oggetto possa essere assegnato a un riferimento di una propria sottoclasse, le operazioni di *casting* di un riferimento possono essere utilizzate per trasformare il riferimento a un oggetto in un riferimento avente come tipo una superclasse – si tratta in questo caso di ***up-casting*** – oppure in un riferimento a una sottoclasse – si tratta allora di un ***down-casting***.

ESEMPIO

Con riferimento alla gerarchia delle classi *Dipendente*, *Impiegato* e *Docente* introdotta nel paragrafo precedente, il seguente frammento di codice esemplifica alcune operazioni di *up-casting* e *down-casting*:

```
...
Dipendente dipendente1, dipendente2;
Impiegato impiegato1, impiegato2;
impiegato1 = new Impiegato("Rossi Mario", 'M', "Via del mare, 1 - Livorno",
                           "Segreteria");
Docente docente1, docente2;
docente1 = new Docente("Neri Maria", 'F', "Via del monte, 99 - Livorno", "Supplente",
                       "Informatica");
...
dipendente1 = (Dipendente)impiegato1;  // up-casting
dipendente2 = (Dipendente)docente1;    // up-casting
impiegato2 = (Impiegato)dipendente1;   // down-casting legale
docente2 = (Docente)dipendente2;       // down-casting legale
impiegato2 = (Impiegato)dipendente2;   // down-casting illegale
docente2 = (Docente)dipendente1;       // down-casting illegale
...
```

Il *casting* dei riferimenti agli oggetti dipende quindi dalle relazioni di ereditarietà che intercorrono tra le classi di una gerarchia.

Qualsiasi riferimento a un oggetto può sempre essere assegnato a un riferimento di tipo *Object* perché la classe *Object* è la superclasse di ogni classe definita nel linguaggio Java.

Quando il *casting* di un riferimento viene effettuato da una sottoclasse a una sua superclasse non è necessario esplicitarlo, perché viene sempre effettuato implicitamente dal compilatore del linguaggio Java.

ESEMPIO

Il metodo *equals* della classe *PuntoOrientato* può essere in realtà codificato in questo modo:

```
public boolean equals(PuntoOrientato po) {
   return (super.equals(po) &&
           getOrientamento() == po.getOrientamento());
}
```

Il *casting* dell'oggetto *po* al tipo *PuntoOrientato* necessario per l'invocazione del metodo *equals* della classe *Punto* viene effettuato implicitamente.

Più in generale, si ha un *casting* implicito quando il riferimento a un oggetto è assegnato a un riferimento il cui tipo:

- è lo stesso della classe di cui l'oggetto è istanza;
- è la classe *Object*;
- è una superclasse della classe di cui l'oggetto è istanza.

Le regole di compilazione non permettono di operare in tutti quei casi in cui il *casting* non è possibile (quando, per esempio, le classi dei riferimenti non hanno tra loro alcuna relazione gerarchica di ereditarietà), ma in ogni caso in fase di esecuzione del programma (*run-time*) si può verificare un'eccezione di tipo *ClassCastException* se l'oggetto del *casting* non è compatibile con il tipo del riferimento a cui viene assegnato.

ESEMPIO

Riprendendo il precedente esempio relativo alla gerarchia delle classi *Dipendente*, *Impiegato* e *Docente*, il seguente frammento di codice esemplifica alcune operazioni di *casting* e il loro esito (per semplicità sono stati omessi i parametri dei costruttori):

```
...
// compilazione corretta: nessun casting
Dipendente dipendente1 = new Dipendente(...);
Docente docente1 = new Docente(...);
Impiegato impiegato1 = new Impiegato(...);
// compilazione corretta: up-casting implicito
Dipendente dipendente2 = new Docente(...);
Dipendente dipendente3 = new Impiegato(...);
Object oggetto1 = docente1;
// compilazione errata: tipi NON compatibili per il casting implicito
Docente docente2 = new Dipendente(...);
Docente docente3 = new Impiegato(...);
Impiegato impiegato2 = new Docente(...);
Impiegato impiegato3 = new Dipendente(...);
// compilazione corretta con errore in fase di esecuzione:
// casting esplicito di tipi incompatibili
Docente docente4 = (Docente)dipendente1; // è un dipendente
Impiegato impiegato4 = (Impiegato)dipendente1; // è un dipendente
Docente docente5 = (Docente)impiegato1; // è un docente
Impiegato impiegato5 = (Impiegato)docente1; // è un docente
Docente docente6 = (Docente)dipendente3; // è un impiegato
Impiegato impiegato6 = (Impiegato)dipendente2; // è un docente
Object oggetto2 = (Impiegato)oggetto1; // è un docente
// compilazione corretta ed esecuzione priva di errori:
// casting esplicito di tipi compatibili
Docente docente7 = (Docente)dipendente2; // è un docente
Impiegato impiegato7 = (Impiegato)dipendente3; // è un impiegato
...
```

Dalla trattazione svolta risulta chiaro che l'ambiente di esecuzione dei programmi in linguaggio Java deve «ricordare» per ogni singolo oggetto la classe da cui è stato istanziato. Senza anticipare il tema del polimorfismo, che è centrale nella programmazione OO, uno dei vantaggi derivanti dall'*up-casting* dei tipi è la possibilità di memorizzare in un'unica struttura dati – come, per esempio, un vettore – oggetti eterogenei istanze di classi diverse della stessa gerarchia di ereditarietà.

ESEMPIO

Dovendo costruire un elenco di dipendenti includendo sia gli impiegati sia i docenti, è sufficiente dichiarare un vettore come il seguente

```
Dipendente elenco[] = new Dipendente[100];
```

ai cui elementi è possibile assegnare sia oggetti di tipo *Impiegato* sia oggetti di tipo *Docente* (oltre che oggetti di tipo *Dipendente* ovviamente):

```
elenco[0] = new Docente(...);
elenco[1] = new Impiegato(...);
```

Nella situazione dell'esempio precedente è importante poter distinguere il tipo dei singoli elementi del vettore per poter effettuare un corretto *down-casting*; questa problematica prende il nome di **RTTI** (***Run-Time Type Identification***) ed è trattata nel seguito del capitolo.

Anche se non è una tecnica molto praticata dai programmatori, è possibile operare l'*overriding* degli attributi di una classe modificandone eventualmente anche il tipo nella classe derivata. In realtà in questo caso, più che di *overriding*, si dovrebbe parlare di *hiding*, in quanto la classe derivata eredita anche gli eventuali attributi sovrascritti che esistono, ma non risultano direttamente accessibili se non ricorrendo alla parola chiave `super` o utilizzando un riferimento del tipo della superclasse (eventualmente ottenuto mediante un *casting*).

ESEMPIO

L'output prodotto dal metodo *main* della classe *Test* esemplifica la situazione di *hiding* dell'attributo *val*:

```
public class Uno {
  public String val = "uno";

  public void show() {
    System.out.println(val);
  }
}

public class Due extends Uno {
  public int val = 2;

  public void show() {
    System.out.println(val);
  }

    public void superShow() {
    System.out.println(super.val);
```

```
        super.show();
    }
}

public class Test {
    public static void main (String[] args) {
        Due due = new Due();
        Uno uno = due;

        uno.show();
        due.show();

        System.out.println(uno.val);
        System.out.println(due.val);
        System.out.println(((Uno)due).val);
        due.superShow();
    }
}
```

```
2
2
uno
2
uno
uno
uno
```

Il *casting* esplicito nell'istruzione *((Uno)due).val* prevede l'accesso all'attributo *val* dell'oggetto *due* riferito come oggetto di classe *Uno*: l'attributo selezionato in questo caso è *val* della classe *Uno*, che in ogni caso l'oggetto *due* eredita anche se nascosto dallo *hiding* operato su tale attributo.

3 La classe *Object* e l'*overriding* del metodo *clone*

La classe *Object*, superclasse di qualsiasi classe definita in un programma in linguaggio Java, definisce – oltre ai metodi *equals* e *toString* che sono stati illustrati nel paragrafo precedente – il metodo ***clone*** la cui firma è

```
protected Object clone() throws CloneNotSupportedException {...}
```

e che consente di effettuare copie di oggetti con una modalità più adatta al contesto dell'ereditarietà rispetto alla tecnica, introdotta nel capitolo A2, del costruttore di copia.

OSSERVAZIONE Si ricordi che il costruttore di copia era stato introdotto perché l'operatore di assegnamento «=» del linguaggio Java non effettua una copia dell'oggetto, ma una duplicazione del riferimento allo stesso oggetto.

Due distinti oggetti istanziati in un programma in linguaggio Java possono essere considerati uguali in base a due diversi criteri:

1. i loro attributi corrispondenti assumono lo stesso valore se sono di tipo primitivo, o riferiscono lo stesso oggetto se sono riferimenti;
2. i loro attributi corrispondenti assumono lo stesso valore se sono di tipo primitivo, o riferiscono oggetti che sono ricorsivamente uguali se sono riferimenti.

I due criteri coincidono nel caso di oggetti istanze di classi che hanno esclusivamente attributi di tipo primitivo.

Dato che un'operazione di copia ha come scopo la creazione di un nuovo oggetto «uguale» a un oggetto preesistente, esistono di conseguenza due diverse tipologie di copia:

- ***shallow copy*** (copia superficiale): viene creato un nuovo oggetto uguale all'originale secondo il criterio di uguaglianza 1: gli attributi del nuovo oggetto creato hanno lo stesso valore dell'oggetto originale, sia che si tratti di tipi primitivi sia di riferimenti;
- ***deep copy*** (copia profonda): viene creato un nuovo oggetto uguale all'originale secondo il criterio di uguaglianza 2: gli attributi del nuovo oggetto, se non sono di tipo primitivo, nel qual caso assumono lo stesso valore, sono ricorsivamente copie profonde degli attributi corrispondenti dell'oggetto originale.

La *deep copy* impone di copiare ogni eventuale oggetto riferito da un attributo e ogni singolo elemento degli eventuali *array*: la sua applicazione ricorsiva comporta che clonare in profondità un oggetto significa clonare tutti gli oggetti raggiungibili a partire dal suo riferimento iniziale. Avendo talvolta a che fare con oggetti complessi si tratta potenzialmente di una tecnica poco efficiente.

Premesso che i due tipi di copia coincidono nel caso di oggetti in cui tutti gli attributi sono di tipo primitivo, se si desidera disporre di un meccanismo di copia in cui l'oggetto creato risulti completamente indipendente in tutte le sue componenti dall'oggetto originale (cioè senza riferimenti comuni), dato che il metodo *clone* ereditato dalla classe *Object* implementa una *shallow copy*, è necessario realizzare la *deep copy* ridefinendo il metodo *clone* stesso.

ESEMPIO Nel seguente metodo *main* viene effettua una *shallow copy* (in questo specifico caso equivalente a una *deep copy*, essendo tutti gli attributi di tipo primitivo) di un oggetto di tipo *PuntoOrientato* invocando il metodo *clone* della classe *Object*:

```
public static void main(String args[])
                    throws CloneNotSupportedException {
  PuntoOrientato po1, po2;

  po1 = new PuntoOrientato(2, 3, 'B');
  po2 = (PuntoOrientato)po1.clone();
}
```

1. i loro attributi corrispondenti assumono lo stesso valore se sono di tipo primitivo, o riferiscono lo stesso oggetto se sono riferimenti;
2. i loro attributi corrispondenti assumono lo stesso valore se sono di tipo primitivo, o riferiscono oggetti che sono ricorsivamente uguali se sono riferimenti.

I due criteri coincidono nel caso di oggetti istanze di classi che hanno esclusivamente attributi di tipo primitivo.

Dato che un'operazione di copia ha come scopo la creazione di un nuovo oggetto «uguale» a un oggetto preesistente, esistono di conseguenza due diverse tipologie di copia:

- ***shallow copy*** (copia superficiale): viene creato un nuovo oggetto uguale all'originale secondo il criterio di uguaglianza 1: gli attributi del nuovo oggetto creato hanno lo stesso valore dell'oggetto originale, sia che si tratti di tipi primitivi sia di riferimenti;
- ***deep copy*** (copia profonda): viene creato un nuovo oggetto uguale all'originale secondo il criterio di uguaglianza 2: gli attributi del nuovo oggetto, se non sono di tipo primitivo, nel qual caso assumono lo stesso valore, sono ricorsivamente copie profonde degli attributi corrispondenti dell'oggetto originale.

La *deep copy* impone di copiare ogni eventuale oggetto riferito da un attributo e ogni singolo elemento degli eventuali *array*: la sua applicazione ricorsiva comporta che clonare in profondità un oggetto significa clonare tutti gli oggetti raggiungibili a partire dal suo riferimento iniziale. Avendo talvolta a che fare con oggetti complessi si tratta potenzialmente di una tecnica poco efficiente.

Premesso che i due tipi di copia coincidono nel caso di oggetti in cui tutti gli attributi sono di tipo primitivo, se si desidera disporre di un meccanismo di copia in cui l'oggetto creato risulti completamente indipendente in tutte le sue componenti dall'oggetto originale (cioè senza riferimenti comuni), dato che il metodo *clone* ereditato dalla classe *Object* implementa una *shallow copy*, è necessario realizzare la *deep copy* ridefinendo il metodo *clone* stesso.

ESEMPIO Nel seguente metodo *main* viene effettua una *shallow copy* (in questo specifico caso equivalente a una *deep copy*, essendo tutti gli attributi di tipo primitivo) di un oggetto di tipo *PuntoOrientato* invocando il metodo *clone* della classe *Object*:

```
public static void main(String args[])
                    throws CloneNotSupportedException {
  PuntoOrientato po1, po2;

  po1 = new PuntoOrientato(2, 3, 'B');
  po2 = (PuntoOrientato)po1.clone();
}
```

Nel codice dell'esempio precedente:

- è dichiarata la possibile generazione di un'eccezione di classe *CloneNotSupportedException* eventualmente sollevata dall'invocazione del metodo *clone*;
- viene effettuato il *casting* al tipo *PuntoOrientato* del risultato dell'invocazione del metodo *clone* che restituisce un oggetto di tipo *Object*.

Entrambi questi aspetti possono essere evitati ridefinendo il metodo *clone* nella classe *PuntoOrientato*:

```
public PuntoOrientato clone() {
  PuntoOrientato po = null;

  try {
    po = new PuntoOrientato(this.x, this.y,
                            this.orientamento);
  }
  catch (EccezionePuntoOrientato eccezione) {
  }

  return po;
}

public static void main(String args[]) {
  PuntoOrientato po1, po2;

  po1 = new PuntoOrientato(2, 3, 'B');
  po2 = po1.clone();
}
```

Utilizzando il costruttore di copia della classe *PuntoOrientato* il metodo *main* dell'esempio precedente sarebbe stato il seguente:

```
public static void main(String args[]) {
  PuntoOrientato po1, po2;

  po1 = new PuntoOrientato (2, 3, 'B');
  po2 = new PuntoOrientato (po1);
}
```

In questo caso l'uso del costruttore di copia è più pratico del ricorso al metodo *clone*, ma, come avremo modo di scoprire in alcune situazioni ricorrenti, il costruttore di copia non può essere invocato.

Affinché gli oggetti istanza di una determinata classe siano clonabili, è necessario che alcune condizioni siano soddisfatte:

- la classe deve implementare l'interfaccia *Cloneable* del *package java.lang*;
- tutti gli oggetti riferiti dagli attributi della classe dell'oggetto da clonare devono essere istanze di classi che implementano a loro volta l'interfaccia *Cloneable*.

Nel codice dell'esempio precedente:

- è dichiarata la possibile generazione di un'eccezione di classe *CloneNotSupportedException* eventualmente sollevata dall'invocazione del metodo *clone*;
- viene effettuato il *casting* al tipo *PuntoOrientato* del risultato dell'invocazione del metodo *clone* che restituisce un oggetto di tipo *Object*.

Entrambi questi aspetti possono essere evitati ridefinendo il metodo *clone* nella classe *PuntoOrientato*:

```
public PuntoOrientato clone() {
  PuntoOrientato po = null;

  try {
    po = new PuntoOrientato(this.x, this.y,
                            this.orientamento);
  }
  catch (EccezionePuntoOrientato eccezione) {
  }

  return po;
}

public static void main(String args[]) {
  PuntoOrientato po1, po2;

  po1 = new PuntoOrientato(2, 3, 'B');
  po2 = po1.clone();
}
```

Utilizzando il costruttore di copia della classe *PuntoOrientato* il metodo *main* dell'esempio precedente sarebbe stato il seguente:

```
public static void main(String args[]) {
  PuntoOrientato po1, po2;

  po1 = new PuntoOrientato (2, 3, 'B');
  po2 = new PuntoOrientato (po1);
}
```

In questo caso l'uso del costruttore di copia è più pratico del ricorso al metodo *clone*, ma, come avremo modo di scoprire in alcune situazioni ricorrenti, il costruttore di copia non può essere invocato.

Affinché gli oggetti istanza di una determinata classe siano clonabili, è necessario che alcune condizioni siano soddisfatte:

- la classe deve implementare l'interfaccia *Cloneable* del *package java.lang*;
- tutti gli oggetti riferiti dagli attributi della classe dell'oggetto da clonare devono essere istanze di classi che implementano a loro volta l'interfaccia *Cloneable*.

ESEMPIO La classe *PuntoOrientato* degli esempi precedenti deve essere così dichiarata:

```
public class PuntoOrientato extends Punto implements Cloneable {...}
```

Dal momento che il metodo *clone* della classe *Object* è definito di livello `protected` può essere invocato solo dai metodi delle classi dello stesso *package* e delle classi direttamente derivate. Per questo motivo dovrebbe essere sempre sovrascritto da un metodo di livello `public`.

ESEMPIO Facendo riferimento alla classe *Mensola* introdotta nel capitolo A3, volendo clonare un oggetto di tipo *Libro* nel codice del metodo *main*, il compilatore impedisce l'invocazione del metodo *clone* ereditato dalla classe *Object*. La soluzione è quella di ridefinire il metodo nella classe *Libro*:

```
public Libro clone() throws CloneNotSupportedException {
  Libro l = null;
  // invocazione metodo clone della classe Object
  l = (Libro)super.clone();
  return l;
}
```

Perché non sia generata un'eccezione di tipo *CloneNotSupportedException* la dichiarazione della classe *Libro* deve essere la seguente:

```
public class Libro implements Cloneable {...}
```

Nel codice dell'esempio precedente si noti che l'uso della parola chiave `super` è di importanza fondamentale, perché specifica l'invocazione del metodo *clone* della classe *Object*. In assenza della parola chiave `super` sarebbe invocato in modo infinitamente ricorsivo il metodo *clone* della classe *Libro*. Il *casting* del risultato al tipo *Libro* è reso necessario dal fatto che il metodo *clone* restituisce un riferimento di tipo *Object*.

ESEMPIO Sempre facendo riferimento alla classe *Mensola* introdotta nel capitolo A3, la *deep copy* degli oggetti istanza della classe richiede, oltre all'implementazione dell'interfaccia *Cloneable*, la definizione del seguente metodo di clonazione, il cui codice scorre l'intero vettore copiandone – mediante invocazione del metodo *clone* della classe *Libro* – i singoli elementi non nulli:

```
public Mensola clone() throws CloneNotSupportedException {
  int posizione;
  Mensola mensola = new Mensola();

  for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
    if (volumi[posizione] != null)
      mensola.setVolume(volumi[posizione].clone(), posizione);

  return mensola;
}
```

ESEMPIO La classe *PuntoOrientato* degli esempi precedenti deve essere così dichiarata:

```
public class PuntoOrientato extends Punto implements Cloneable {...}
```

Dal momento che il metodo *clone* della classe *Object* è definito di livello `protected` può essere invocato solo dai metodi delle classi dello stesso *package* e delle classi direttamente derivate. Per questo motivo dovrebbe essere sempre sovrascritto da un metodo di livello `public`.

ESEMPIO Facendo riferimento alla classe *Mensola* introdotta nel capitolo A3, volendo clonare un oggetto di tipo *Libro* nel codice del metodo *main*, il compilatore impedisce l'invocazione del metodo *clone* ereditato dalla classe *Object*. La soluzione è quella di ridefinire il metodo nella classe *Libro*:

```
public Libro clone() throws CloneNotSupportedException {
  Libro l = null;
  // invocazione metodo clone della classe Object
  l = (Libro)super.clone();
  return l;
}
```

Perché non sia generata un'eccezione di tipo *CloneNotSupportedException* la dichiarazione della classe *Libro* deve essere la seguente:

```
public class Libro implements Cloneable {...}
```

Nel codice dell'esempio precedente si noti che l'uso della parola chiave `super` è di importanza fondamentale, perché specifica l'invocazione del metodo *clone* della classe *Object*. In assenza della parola chiave `super` sarebbe invocato in modo infinitamente ricorsivo il metodo *clone* della classe *Libro*. Il *casting* del risultato al tipo *Libro* è reso necessario dal fatto che il metodo *clone* restituisce un riferimento di tipo *Object*.

ESEMPIO Sempre facendo riferimento alla classe *Mensola* introdotta nel capitolo A3, la *deep copy* degli oggetti istanza della classe richiede, oltre all'implementazione dell'interfaccia *Cloneable*, la definizione del seguente metodo di clonazione, il cui codice scorre l'intero vettore copiandone – mediante invocazione del metodo *clone* della classe *Libro* – i singoli elementi non nulli:

```
public Mensola clone() throws CloneNotSupportedException {
  int posizione;
  Mensola mensola = new Mensola();

  for (posizione=0; posizione<MAX_NUM_VOLUMI; posizione++)
    if (volumi[posizione] != null)
      mensola.setVolume(volumi[posizione].clone(), posizione);

  return mensola;
}
```

Dopo aver eseguito una *deep copy* di un oggetto di classe *Mensola* invocandone il metodo *clone*, la situazione è quella schematizzata in FIGURA 4.

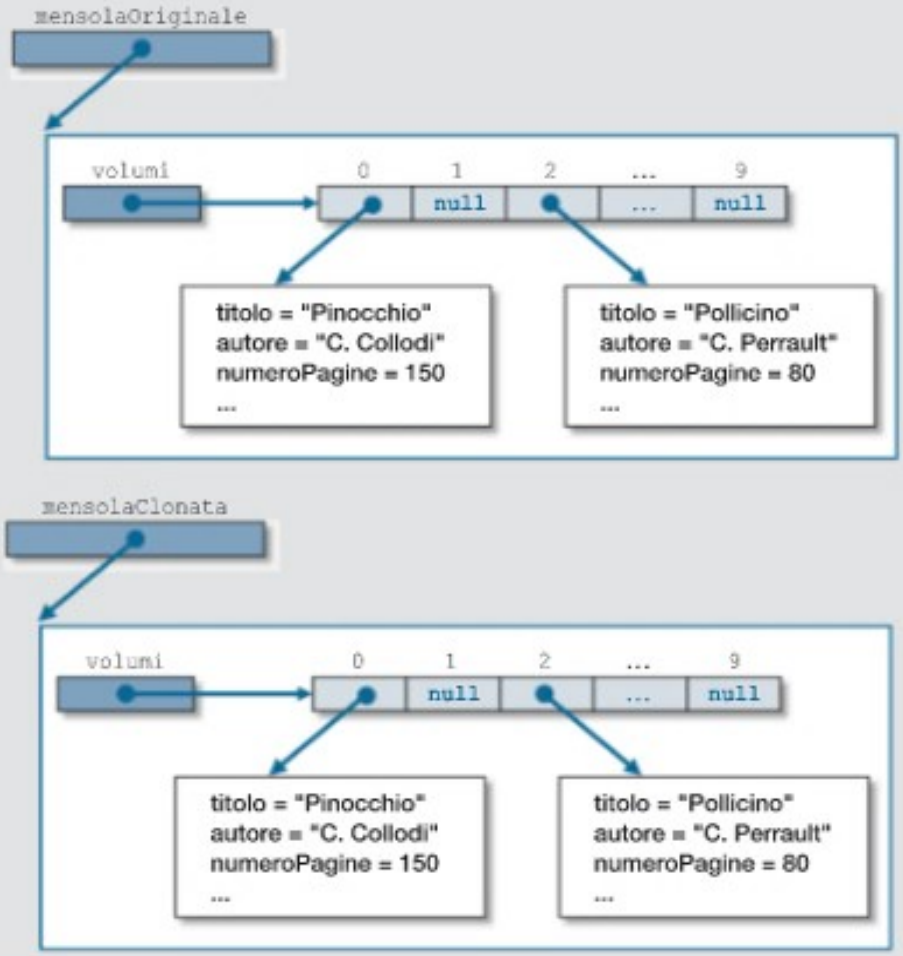


FIGURA 4

Se invece si fosse effettuata una *shallow copy* definendo il seguente metodo per la clonazione degli oggetti

```
public Mensola clone() throws CloneNotSupportedException {
  return (Mensola)super.clone();
}
```

l'oggetto restituito avrebbe condiviso con quello originale lo stesso *array*: aggiungendo o eliminando libri da una delle due mensole la modifica si sarebbe automaticamente propagata anche all'altra (FIGURA 5)!

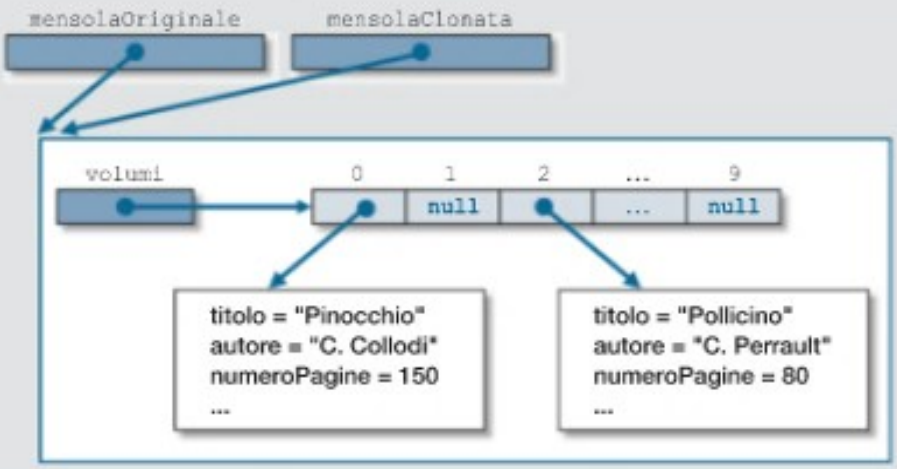


FIGURA 5

Dopo aver eseguito una *deep copy* di un oggetto di classe *Mensola* invocandone il metodo *clone*, la situazione è quella schematizzata in FIGURA 4.

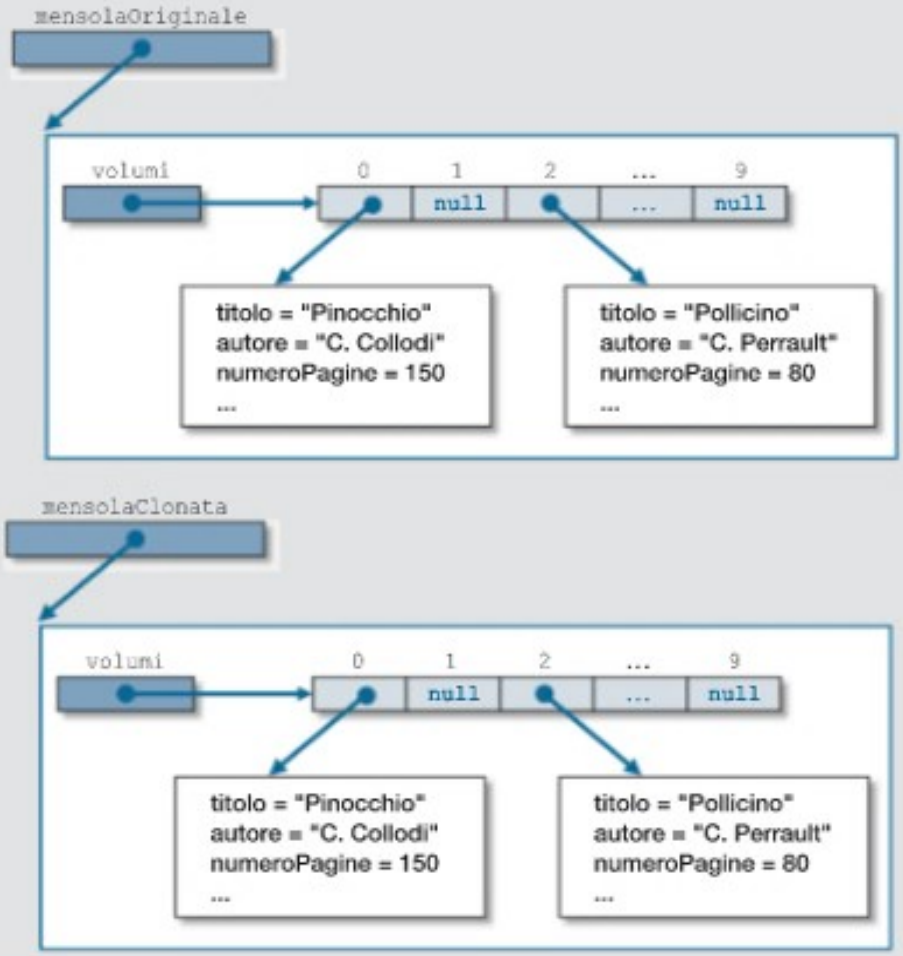


FIGURA 4

Se invece si fosse effettuata una *shallow copy* definendo il seguente metodo per la clonazione degli oggetti

```
public Mensola clone() throws CloneNotSupportedException {
  return (Mensola)super.clone();
}
```

l'oggetto restituito avrebbe condiviso con quello originale lo stesso *array*: aggiungendo o eliminando libri da una delle due mensole la modifica si sarebbe automaticamente propagata anche all'altra (FIGURA 5)!

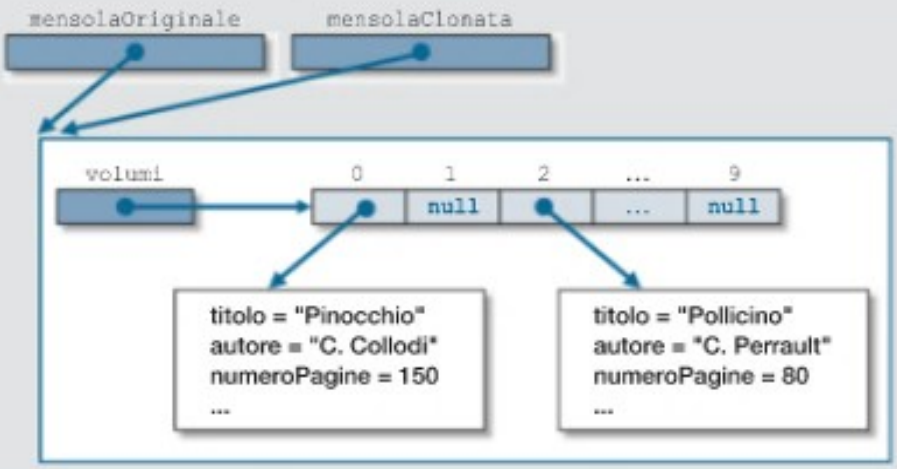


FIGURA 5

Il meccanismo di serializzazione degli oggetti trattato nel capitolo A3 consente di trasformare le strutture complesse degli oggetti in forma di flussi di byte e di crearne copie equivalenti. La serializzazione costituisce quindi un ulteriore modo di ottenere copie di oggetti in Java.

4 Classi astratte e interfacce

Ritornando all'esempio della gerarchia di classi *Dipendente*, *Impiegato* e *Docente* introdotta nei paragrafi precedenti, si osserva che probabilmente non vi è nessun interesse a creare oggetti di classe *Dipendente*, ma solo a istanziare impiegati e docenti, che sono anche dipendenti (e da questo punto di vista possono essere trattati in modo uniforme).

La parola chiave **abstract** del linguaggio Java premessa alla dichiarazione di una classe fa in modo che non possano essere istanziati oggetti aventi come tipo la classe stessa: è evidente che una **classe astratta** viene realizzata all'unico scopo di poter derivare da essa una o più classi **concrete**. Il diagramma UML di una classe astratta riporta il nome della classe in *carattere corsivo*.

ESEMPIO

La classe *Dipendente* dovrebbe essere così dichiarata:

```
abstract public class Dipendente {...}
```

In questo modo dichiarazioni come la seguente risultano illegali e generano un errore a tempo di compilazione:

```
...
Dipendente dipendente = new Dipendente("Rossi Mario", 'M',
                                       "Via del mare, 1");
...
```

A partire da una classe astratta non si possono istanziare oggetti, ma è possibile dichiarare riferimenti a cui assegnare oggetti istanza di classi derivate, come nel seguente frammento di codice:

```
...
Dipendente dipendente1, dipendente2;
dipendente1 = new Impiegato("Rossi Mario", 'M',
                            "Via del mare, 1 - Livorno",
                            "Segreteria");
dipendente2 = new Docente("Neri Maria", 'F',
                          "Via del monte, 99 - Livorno",
                          "Supplente", "Informatica");
...
```

4.1 Classi astratte e metodi astratti

Spesso lo scopo di una gerarchia di classi è fattorizzare attributi comuni a più classi al livello più alto della gerarchia, in modo tale che siano ereditati dalle classi di livello più basso. In questo contesto le classi astratte consen-

Il meccanismo di serializzazione degli oggetti trattato nel capitolo A3 consente di trasformare le strutture complesse degli oggetti in forma di flussi di byte e di crearne copie equivalenti. La serializzazione costituisce quindi un ulteriore modo di ottenere copie di oggetti in Java.

4 Classi astratte e interfacce

Ritornando all'esempio della gerarchia di classi *Dipendente*, *Impiegato* e *Docente* introdotta nei paragrafi precedenti, si osserva che probabilmente non vi è nessun interesse a creare oggetti di classe *Dipendente*, ma solo a istanziare impiegati e docenti, che sono anche dipendenti (e da questo punto di vista possono essere trattati in modo uniforme).

La parola chiave **abstract** del linguaggio Java premessa alla dichiarazione di una classe fa in modo che non possano essere istanziati oggetti aventi come tipo la classe stessa: è evidente che una **classe astratta** viene realizzata all'unico scopo di poter derivare da essa una o più classi **concrete**. Il diagramma UML di una classe astratta riporta il nome della classe in *carattere corsivo*.

ESEMPIO

La classe *Dipendente* dovrebbe essere così dichiarata:

```
abstract public class Dipendente {...}
```

In questo modo dichiarazioni come la seguente risultano illegali e generano un errore a tempo di compilazione:

```
...
Dipendente dipendente = new Dipendente("Rossi Mario", 'M',
                                       "Via del mare, 1");
...
```

A partire da una classe astratta non si possono istanziare oggetti, ma è possibile dichiarare riferimenti a cui assegnare oggetti istanza di classi derivate, come nel seguente frammento di codice:

```
...
Dipendente dipendente1, dipendente2;
dipendente1 = new Impiegato("Rossi Mario", 'M',
                            "Via del mare, 1 - Livorno",
                            "Segreteria");
dipendente2 = new Docente("Neri Maria", 'F',
                          "Via del monte, 99 - Livorno",
                          "Supplente", "Informatica");
...
```

4.1 Classi astratte e metodi astratti

Spesso lo scopo di una gerarchia di classi è fattorizzare attributi comuni a più classi al livello più alto della gerarchia, in modo tale che siano ereditati dalle classi di livello più basso. In questo contesto le classi astratte consen-

tono di usare questo meccanismo di fattorizzazione degli attributi comuni anche nel caso in cui la superclasse risultante non sia «ben definita».

L'uso delle classi astratte nel contesto del linguaggio di programmazione Java è disciplinato dalle seguenti regole fondamentali:

- una classe viene definita astratta premettendo alla sua dichiarazione il qualificatore `abstract`;
- una classe astratta può avere costruttori, ma **non** può essere istanziata;
- una classe astratta può avere uno o più «metodi astratti», cioè metodi privi del corpo e qualificati con la parola chiave `abstract` la cui implementazione è vincolante per le classi derivate;
- la dichiarazione di uno o più metodi astratti rende implicitamente la classe astratta;
- se una classe concreta deriva da una classe astratta **deve** fornire una implementazione per tutti i metodi astratti ereditati;
- le classi astratte possono essere superclassi o sottoclassi di altre classi, sia astratte sia non.

ESEMPIO

Le seguenti classi *Sfera* e *Cubo* hanno alcuni attributi e metodi comuni, anche se il codice dei metodi comuni è diverso:

```
public class Sfera {
   private double raggio;
   private double pesoSpecifico;

   public Sfera(double raggio, double pesoSpecifico) {
      this.raggio = raggio;
      this.pesoSpecifico = pesoSpecifico;
   }

   public double getRaggio() {
      return raggio;
   }

   public double getPesoSpecifico() {
      return pesoSpecifico;
   }

   public double volume() {
      return 4/3 * Math.PI * Math.pow(raggio,3);
   }

   public double superficie() {
      return 4 * Math.PI * Math.pow(raggio,2);
   }

   public double peso() {
      return pesoSpecifico * volume();
   }
}

public class Cubo {
   private double lato;
```

tono di usare questo meccanismo di fattorizzazione degli attributi comuni anche nel caso in cui la superclasse risultante non sia «ben definita».

L'uso delle classi astratte nel contesto del linguaggio di programmazione Java è disciplinato dalle seguenti regole fondamentali:

- una classe viene definita astratta premettendo alla sua dichiarazione il qualificatore `abstract`;
- una classe astratta può avere costruttori, ma **non** può essere istanziata;
- una classe astratta può avere uno o più «metodi astratti», cioè metodi privi del corpo e qualificati con la parola chiave `abstract` la cui implementazione è vincolante per le classi derivate;
- la dichiarazione di uno o più metodi astratti rende implicitamente la classe astratta;
- se una classe concreta deriva da una classe astratta **deve** fornire una implementazione per tutti i metodi astratti ereditati;
- le classi astratte possono essere superclassi o sottoclassi di altre classi, sia astratte sia non.

ESEMPIO

Le seguenti classi *Sfera* e *Cubo* hanno alcuni attributi e metodi comuni, anche se il codice dei metodi comuni è diverso:

```
public class Sfera {
   private double raggio;
   private double pesoSpecifico;

   public Sfera(double raggio, double pesoSpecifico) {
      this.raggio = raggio;
      this.pesoSpecifico = pesoSpecifico;
   }

   public double getRaggio() {
      return raggio;
   }

   public double getPesoSpecifico() {
      return pesoSpecifico;
   }

   public double volume() {
      return 4/3 * Math.PI * Math.pow(raggio,3);
   }

   public double superficie() {
      return 4 * Math.PI * Math.pow(raggio,2);
   }

   public double peso() {
      return pesoSpecifico * volume();
   }
}

public class Cubo {
   private double lato;
```

```
    private double pesoSpecifico;

    public Cubo(double lato, double pesoSpecifico) {
        this.lato = lato;
        this.pesoSpecifico = pesoSpecifico;
    }

    public double getLato() {
        return lato;
    }

    public double getPesoSpecifico() {
        return pesoSpecifico;
    }

    public double volume() {
        return Math.pow(lato,3);
    }

    public double superficie() {
        return 6 * Math.pow(lato,2);
    }

    public double peso() {
        return pesoSpecifico * volume();
    }
}
```

La classe astratta *Solido* fattorizza attributi e metodi comuni alle classi *Sfera* e *Cubo* (FIGURA **6**):

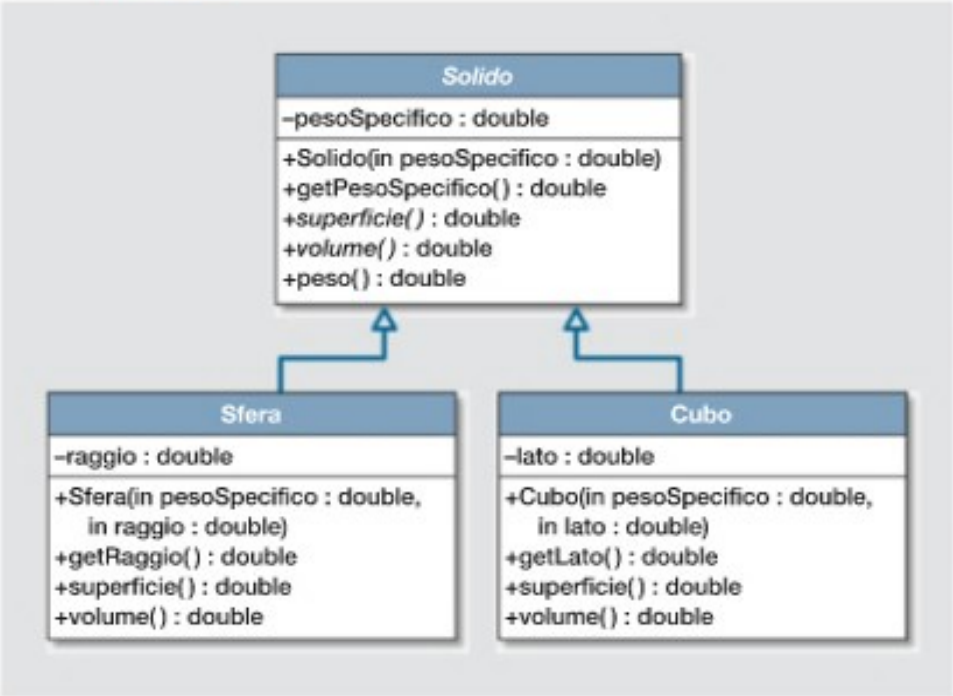


FIGURA **6**

```
abstract public class Solido {
    private double pesoSpecifico;

    public Solido(double pesoSpecifico) {
```

```
    private double pesoSpecifico;

    public Cubo(double lato, double pesoSpecifico) {
        this.lato = lato;
        this.pesoSpecifico = pesoSpecifico;
    }

    public double getLato() {
        return lato;
    }

    public double getPesoSpecifico() {
        return pesoSpecifico;
    }

    public double volume() {
        return Math.pow(lato,3);
    }

    public double superficie() {
        return 6 * Math.pow(lato,2);
    }

    public double peso() {
        return pesoSpecifico * volume();
    }
}
```

La classe astratta *Solido* fattorizza attributi e metodi comuni alle classi *Sfera* e *Cubo* (FIGURA **6**):

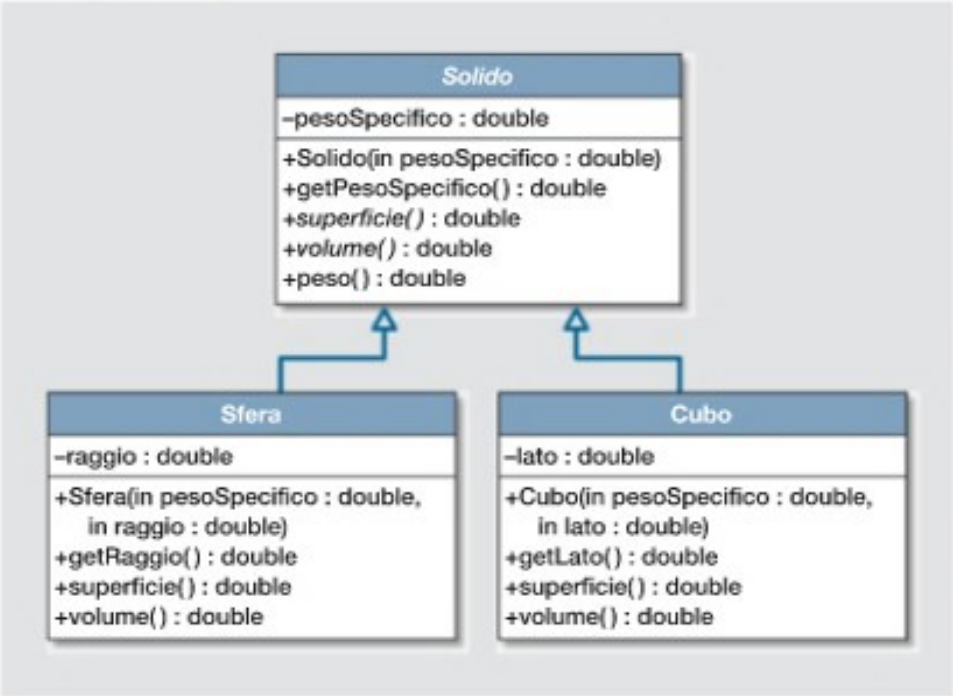


FIGURA **6**

```
abstract public class Solido {
    private double pesoSpecifico;

    public Solido(double pesoSpecifico) {
```

```
      this.pesoSpecifico = pesoSpecifico;
    }

    public double getPesoSpecifico() {
      return pesoSpecifico;
    }

    abstract public double volume();

    abstract public double superficie();

    public double peso() {
      return pesoSpecifico * volume();
    }
  }

  public class Sfera extends Solido {
    private double raggio;

    public Sfera(double raggio, double pesoSpecifico) {
      super(pesoSpecifico);
      this.raggio = raggio;
    }

    public double getRaggio() {
      return raggio;
    }

    public double volume() {
      return 4/3 * Math.PI * Math.pow(raggio,3);
    }

    public double superficie() {
      return 4 * Math.PI * Math.pow(raggio,2);
    }
  }

  public class Cubo extends Solido {
    private double lato;

    public Cubo(double lato, double pesoSpecifico) {
      super(pesoSpecifico);
      this.lato = lato;
    }

    public double getLato() {
      return lato;
    }

    public double volume() {
      return Math.pow(lato,3);
    }

    public double superficie() {
      return 6 * Math.pow(lato,2);
    }
  }
```

```
      this.pesoSpecifico = pesoSpecifico;
    }

    public double getPesoSpecifico() {
      return pesoSpecifico;
    }

    abstract public double volume();

    abstract public double superficie();

    public double peso() {
      return pesoSpecifico * volume();
    }
  }

  public class Sfera extends Solido {
    private double raggio;

    public Sfera(double raggio, double pesoSpecifico) {
      super(pesoSpecifico);
      this.raggio = raggio;
    }

    public double getRaggio() {
      return raggio;
    }

    public double volume() {
      return 4/3 * Math.PI * Math.pow(raggio,3);
    }

    public double superficie() {
      return 4 * Math.PI * Math.pow(raggio,2);
    }
  }

  public class Cubo extends Solido {
    private double lato;

    public Cubo(double lato, double pesoSpecifico) {
      super(pesoSpecifico);
      this.lato = lato;
    }

    public double getLato() {
      return lato;
    }

    public double volume() {
      return Math.pow(lato,3);
    }

    public double superficie() {
      return 6 * Math.pow(lato,2);
    }
  }
```

I metodi *superficie* e *volume* hanno la stessa firma in entrambe le classi *Sfera* e *Cubo*, ma implementazioni diverse: la loro firma è stata fattorizzata nella superclasse *Solido*, priva del corpo del metodo premettendo la parola chiave `abstract` e vincolando in questo modo la loro definizione nelle classi derivate.

4.2 Interfacce

Nel linguaggio di programmazione Java la parola chiave `interface` consente di dichiarare un'**interfaccia** che è analoga a una classe, ma a differenza di questa costituisce una pura specifica di invocazione e come tale si limita a dichiarare i metodi, definendone la firma, senza implementarli (tutti i metodi di un'interfaccia sono implicitamente astratti).

Inoltre, a differenza di una classe astratta, in un'interfaccia è possibile definire attributi costanti, ma non variabili. Nella sua forma più comune un'interfaccia è un elenco di dichiarazioni di metodi privi del corpo.

Un'interfaccia viene **implementata** dalle classi mediante l'uso della parola chiave `implements`.

Il linguaggio di programmazione Java definisce delle interfacce vuote (come, per esempio, *Serializable* e *Cloneable*) al solo scopo di essere utilizzate come «marcatori» per le classi che implementano determinate funzionalità.

Implementare un'interfaccia impone formalmente a una classe di definire i metodi che l'interfaccia stessa dichiara: in questo senso le interfacce costituiscono una specie di «contratto» per le classi che le implementano e il rispetto di tale contratto viene verificato dal compilatore che genera un errore nel caso in cui una classe che dichiara di implementare una determinata interfaccia non ne definisca tutti i metodi dichiarati.

ESEMPIO Con riferimento al precedente esempio relativo alle classi *Solido*, *Sfera* e *Cubo*, è ragionevole pensare che la classe astratta *Solido*, che integra le caratteristiche di una figura solida geometrica e quelle di un corpo solido fisico, implementi le seguenti interfacce:

```
public interface FiguraSolida {
  public double superficie();
  public double volume();
}

public interface CorpoSolido {
  public double peso();
}

abstract public class Solido implements FiguraSolida,
                        CorpoSolido {
```

```
    private double pesoSpecifico;

    public Solido(double pesoSpecifico) {
      this.pesoSpecifico = pesoSpecifico;
    }

    public double getPesoSpecifico() {
      return pesoSpecifico;
    }

    abstract public double volume();

    abstract public double superficie();

    public double peso() {
    return pesoSpecifico * volume();
    }
}
```

Il diagramma UML delle classi assume l'aspetto di FIGURA 7, in cui le interfacce sono caratterizzate dal qualificatore `«interface»`.

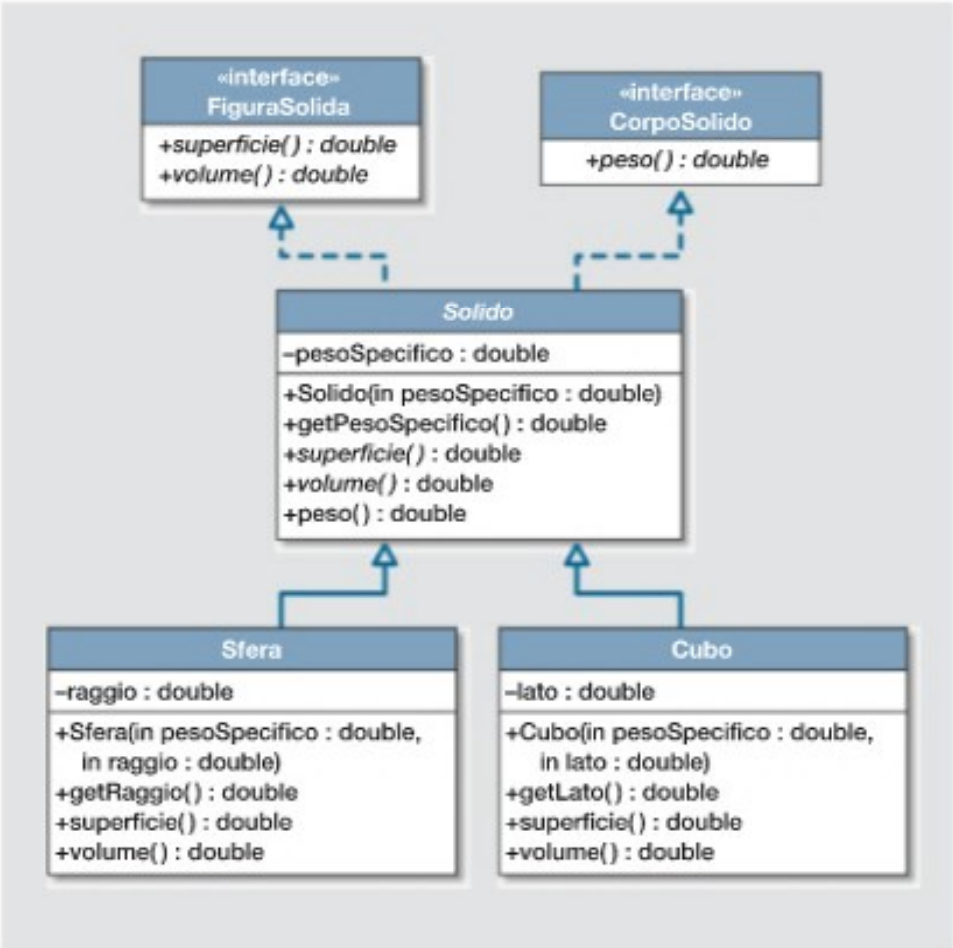


FIGURA 7

Nell'esempio precedente si noti come la classe astratta *Solido* implementi due distinte interfacce: il linguaggio Java consente l'implementazione multipla di interfacce anche se proibisce l'ereditarietà multipla di classi.

Simulare l'ereditarietà multipla in Java

Diversamente, per esempio, dal linguaggio C++, Java non consente a una classe di avere più di una superclasse diretta. Questa soluzione permette di evitare molte problematiche sia di tipo implementativo sia di programmazione, ma pone una severa limitazione al riuso del codice e alla flessibilità di progettazione delle gerarchie delle classi.
Pur non colmando appieno questa mancanza, le interfacce – che possono essere implementate in modo multiplo da una stessa classe – consentono ai programmatori Java di simulare l'ereditarietà multipla.

ESEMPIO Una classe può implementare più interfacce integrando schemi di invocazione diversi; la seguente dichiarazione della classe *Libro*

```
public class Libro implements Serializable, Cloneable {
...
}
```

permette di dotare la stessa sia della possibilità di essere clonabile, sia di quella di essere serializzabile.

Le interfacce possono essere estese per ereditarietà ed è quindi possibile costruire gerarchie di interfacce.

ESEMPIO Il *Collection framework* del linguaggio Java realizza una gerarchia di interfacce implementate dalle classi concrete direttamente utilizzabili dal programmatore (FIGURA **8**).

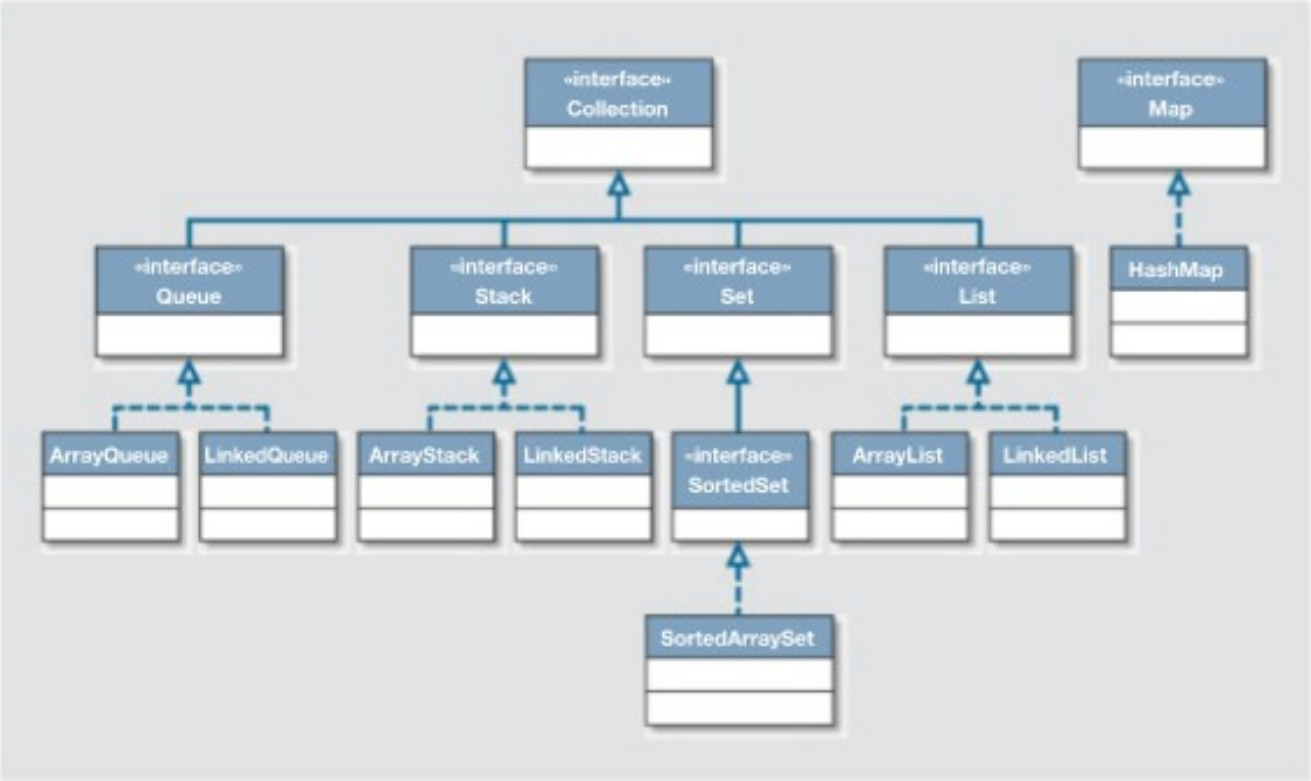


FIGURA **8**

Se un'interfaccia definisce attributi, essi sono implicitamente costanti ed ereditati dalle classi che la implementano (e se qualificati `static` sono ereditati a livello di classe).

ESEMPIO Questa caratteristica viene talvolta utilizzata per aggregare più costanti da utilizzare in classi diverse:

```
interface CostantiFondamentali {
   static final double G = 6.670E-11;    // costante di gravitazione
   static final double c = 2.9979246E8;  // velocità della luce
   static final double N = 6.02252E23;   // numero di Avogadro
   static final double k = 1.38054E23;   // costante di Boltzmann
   static final double R = 8.3143;       // costante dei gas perfetti
   static final double h = 6.62559E-34;  // costante di Planck
}
```

5 Polimorfismo e *binding* dinamico

L'operazione di estensione di una classe permette di derivare da essa una o più sottoclassi: è sempre possibile utilizzare un oggetto istanza di una delle sottoclassi derivate in un contesto in cui sia richiesto un oggetto istanza della classe originale. Questa caratteristica dei linguaggi di programmazione OO in generale e del linguaggio Java in particolare viene detta **polimorfismo**, intendendo con questo termine la capacità di un oggetto di assumere «forme» molteplici: istanza della propria classe o istanza di ogni superclasse di essa.

Il polimorfismo degli oggetti risulta particolarmente utile quando, in una gerarchia di classi derivate, viene ridefinito un metodo e la specifica versione del metodo da eseguire viene scelta automaticamente sulla base del tipo di oggetto effettivamente assegnato a un riferimento in fase di esecuzione, piuttosto che al momento della compilazione.

ESEMPIO

Nella classe *Punto* introdotta nei paragrafi precedenti è definito il metodo *toString*, che viene ridefinito nella classe *PuntoOrientato* sua derivata. Nel codice che segue

```
...
Punto tmp;

if (Math.random() > 0.5)
  tmp = new Punto(1.0, 0.0);
else
  tmp = new PuntoOrientato(1.0, 0.0, 'A');

tmp.toString();
...
```

dato che l'invocazione del metodo statico *random* della classe *Math* genera un numero casuale compreso tra 0 e 1, il compilatore non può determinare se al riferimento *tmp* sarà assegnata un'istanza della classe *Punto* o della classe *PuntoOrientato*. L'ambiente di esecuzione dei programmi Java dovrà determinare il metodo *toString* corretto da invocare (quello definito dalla classe *Punto* piuttosto che quello ridefinito dalla classe *PuntoOrientato*) in fase di esecuzione in base al tipo effettivo di oggetto assegnato al riferimento.

I metodi ridefiniti in una classe derivata sono detti **polimorfi**, in quanto lo stesso metodo ha comportamenti diversificati a seconda del tipo di oggetto su cui è invocato.

Il meccanismo che determina quale metodo deve essere invocato in base alla classe di appartenenza dell'oggetto è definito ***binding*** (letteralmente «legame», «collegamento»). Esistono due tipi fondamentali di *binding*:

- ***binding statico*** (*early binding*), in cui il metodo da invocare viene determinato in fase di compilazione del codice (*compile-time binding*);
- ***binding dinamico*** (*late binding*), in cui il metodo da invocare viene determinato durante l'esecuzione del codice (*run-time binding*).

Il linguaggio Java, come la maggior parte dei linguaggi di programmazione a oggetti, adotta il *binding* dinamico a *run-time* quando il *binding* statico a *compile-time* risulta impossibile, come nel caso dei metodi polimorfici.

ESEMPIO

Con riferimento alla gerarchia di classi *Solido*, *Sfera* e *Cubo*, il seguente metodo *main* di un'ipotetica classe di test

```
public static void main(String[] args) {
   Sfera sfera = new Sfera(1.0,0.5);
   Cubo cubo = new Cubo(3.0,1.1);

   System.out.println("Sfera sfera---> Volume….: " + sfera.volume());
   System.out.println("Sfera sfera---> Peso……: " + sfera.peso());
   System.out.println("Cubo cubo----> Volume….: " + cubo.volume());
   System.out.println("Cubo cubo----> Peso……: " + cubo.peso());
   Solido solido1 = sfera;
   System.out.println("Solido solido1-> Volume….: " + solido1.volume());
   System.out.println("Solido solido1-> Peso……: " + solido1.peso());
   Solido solido2 = cubo;
   System.out.println("Solido solido2-> Volume….: " + solido2.volume());
   System.out.println("Solido solido2-> Peso……: " + solido2.peso());
}
```

produce il seguente output:

```
Sfera sfera---> Volume….: 3.141592653589793
Sfera sfera---> Peso……: 1.5707963267948966
Cubo cubo----> Volume….: 27.0
Cubo cubo----> Peso……: 29.700000000000003
Solido solido1-> Volume….: 3.141592653589793
Solido solido1-> Peso……: 1.5707963267948966
Solido solido2-> Volume….: 27.0Solido solido2-> Peso……: 29.700000000000003
```

Le prime quattro righe visualizzate sono relative all'invocazione dei metodi della classi *Sfera* e *Cubo*, che non sono polimorfici: il metodo *peso* è ereditato dalla classe astratta *Solido* senza essere ridefinito, mentre il metodo *volume* è un'implementazione di un metodo astratto della classe *Solido*.
Le ultime quattro righe visualizzate sono il risultato di un classico esempio di invocazione di metodi polimorfici: ai riferimenti *solido1* e *solido2* sono infatti assegnati rispettivamente gli oggetti *sfera* e *cubo* istanze di classi diverse.
L'invocazione del metodo *volume* comporta il *binding* dinamico a *run-time* che esegue il codice del metodo definito nella classe *Sfera* nel primo caso e nella classe *Cubo* nel secondo caso; l'invocazione del metodo *peso*, invece, esegue il codice dell'unica definizione del metodo data nella classe astratta *Solido* da cui *Sfera* e *Cubo* derivano.

Il polimorfismo consente di sviluppare codice a oggetti **estensibile**: l'aggiunta di nuove classi a una gerarchia preesistente non influenza il codice che gestisce oggetti istanza delle classi della gerarchia stessa in modo polimorfico.

ESEMPIO

L'aggiunta della seguente classe *Cilindro* alla gerarchia delle classi che ereditano dalla classe astratta *Solido*

```
public class Cilindro extends Solido {
   private double raggio;
   private double altezza;

   public Cilindro(double raggio, double altezza,
                   double pesoSpecifico) {
      super(pesoSpecifico);
      this.raggio = raggio;
      this.altezza = altezza;
   }
```

```
    public double getRaggio() {
      return raggio;
    }

    public double getAltezza() {
      return altezza;
    }

    public double volume() {
      return Math.PI * Math.pow(raggio,2) * altezza;
    }

    public double superficie() {
      return Math.PI * Math.pow(raggio,2);
    }
  }
```

non comporta alcuna variazione nel seguente frammento di codice che calcola la media dei volumi e dei pesi di tutti i solidi contenuti in un vettore:

```
...
Solido solidi[] = new Solido[...];
double volumeMedio = 0.0;
double pesoMedio = 0.0;
...
for (int i=0; i<solidi.length; i++) {
  volumeMedio += solidi[i].volume();
  pesoMedio += solidi[i].peso();
}
volumeMedio /= solidi.length;
pesoMedio /= solidi.length;
...
```

La classificazione del polimorfismo

Nel 1985 Luca Cardelli e Peter Wegner classificarono le forme di polimorfismo di un linguaggio di programmazione in due categorie: **universale** e ***ad hoc***.

Nella prima categoria rientrano il *polimorfismo parametrico*, in cui funzioni e operatori sono parametrizzati in base al tipo a cui possono essere applicati (Java realizza questa forma di polimorfismo con le classi «generiche»), e il *polimorfismo per inclusione*, che è esattamente quello implementato dai linguaggi OO: lo stesso metodo può essere applicato a tutti gli oggetti «inclusi» nella classe che lo definisce (cioè istanze di classi derivate).

Nella seconda categoria rientra l'*overloading* delle funzioni e dei metodi in base al tipo dei parametri e del risultato e la cosiddetta *coercion* realizzata dal *casting* implicito dei tipi degli operandi di un operatore, o dei parametri di una funzione o di un metodo.

In assenza di polimorfismo il codice dell'esempio precedente avrebbe dovuto prevedere nel corpo del ciclo un'istruzione di selezione della classe con relativa differenziazione dell'invocazione del metodo *volume*:

```
...
for (int i=0; i<solidi.length; i++) {
  if (solidi[i] instanceof Sfera)
    volumeMedio += (Sfera)solidi[i].volume();
  if (solidi[i] instanceof Cubo)
    volumeMedio += (Cubo)solidi[i].volume();
  if (solidi[i] instanceof Cilindro)
    volumeMedio += (Cilindro)solidi[i].volume();
  pesoMedio += solidi[i].peso();
}
volumeMedio /= solidi.length;
pesoMedio /= solidi.length;
...
```

L'operatore Java **instanceof**, che sarà introdotto nel prossimo paragrafo, consente di verificare la classe di cui un oggetto è istanza. È evidente che senza polimorfismo il codice che invoca il metodo *volume* dovrebbe essere modificato per ogni nuova classe aggiunta alla gerarchia.

Il polimorfismo – nella forma in cui è concretizzato dal linguaggio Java – realizza un equilibrio fra due opposte esigenze:

- il rigido controllo statico dei tipi degli oggetti che consente di riscontrare il maggior numero possibile di errori di programmazione in fase di compilazione;
- il rilassamento della regola che vuole che ogni oggetto abbia un tipo predefinito in fase di compilazione allo scopo di garantire al programmatore una flessibilità adeguata.

Alcuni linguaggi – come, per esempio, il linguaggio JavaScript – eliminano alla radice il problema non imponendo che le variabili abbiano un tipo e non effettuandone di conseguenza il controllo statico. In questo caso tutti i controlli sono effettuati a tempo di esecuzione: lo svantaggio di questa soluzione è che eventuali errori dovuti a tipi incompatibili possono comparire anche molto tempo dopo che un programma è stato rilasciato.

Le problematiche derivanti dalla necessità di duplicare gli oggetti nella memoria *heap* senza avere riferimenti multipli allo stesso oggetto sono ovviamente presenti anche negli oggetti polimorfici; inoltre, nel codice di metodi che restituiscono o ricevono come parametri oggetti polimorfici, non è possibile invocare il costruttore di copia, perché non è nota a priori la classe di cui un oggetto è istanza.

La soluzione più adatta per ovviare a questo problema è quella di definire in tutte le classi di una gerarchia uno specifico metodo *clone* da invocare al posto del costruttore di copia: trattandosi di un metodo polimorfico, sarà automaticamente selezionato il metodo definito nella classe di cui l'oggetto su cui lo si applica è istanza, anche se il riferimento assume il tipo di una superclasse, eventualmente astratta.

6 *Run-Time Type Identification* e operatore *instanceof*

L'esigenza di meccanismi di ***Run-Time Type Identification*** (**RTTI**), ovvero di supporto per l'identificazione del tipo effettivo di un oggetto durante l'esecuzione di un programma, nei linguaggi OO in generale e in Java in particolare, nasce dall'impossibilità di invocare direttamente il metodo di una sottoclasse su un riferimento di una superclasse, cosa che può risultare utile o anche necessaria. Per applicare propriamente il *casting* necessario in questa situazione è indispensabile conoscere il tipo effettivo di cui l'oggetto riferito è istanza.

L'operatore Java `instanceof` permette di verificare la classe di appartenenza di un oggetto.

La «riflessione» in Java

Il linguaggio Java consente ai programmi di creare, ispezionare e gestire le classi a tempo di esecuzione: questa caratteristica del linguaggio è nota con il nome di **riflessione**. In particolare la classe *Class* – le cui possibili istanze sono le classi che costituiscono il programma in esecuzione – rende disponibili metodi per verificare dinamicamente la classe di un oggetto; infatti *Class* eredita da *Object* il metodo *getClass* che restituisce la classe di un oggetto.
Il seguente frammento di codice, per esempio, visualizza il nome della classe dell'oggetto che lo esegue:

```
...
Class c =
this.getClass();
System.out.println
(c.getName());
...
```

Il metodo *getSuperclass* restituisce il tipo della superclasse; il seguente frammento di codice visualizza il nome della superclasse della classe di appartenenza dell'oggetto che lo esegue:

```
...
Class c =
this.getSuperclass();
System.out.println
(c.getName());
...
```

La classe *Class* definisce inoltre il metodo *isInstance* che consente di verificare dinamicamente a *run-time* la classe di appartenenza di un oggetto.

ESEMPIO Facendo riferimento alla gerarchia di classi *Dipendente*, *Impiegato* e *Docente*, può essere necessario invocare il metodo *setUfficio* per un oggetto di classe *Impiegato* assegnato a un riferimento di tipo *Dipendente* mediante un *casting*:

```
...
((Impiegato)dipendente).setUfficio("Personale");
...
```

Naturalmente, nel caso in cui l'oggetto assegnato al riferimento *dipendente* non sia un'istanza della classe *Impiegato*, viene generata un'eccezione; per evitare l'errore si può condizionare l'istruzione a una verifica della classe di appartenenza dell'oggetto assegnato al riferimento:

```
...
if (dipendente instanceof Impiegato)
  ((Impiegato)dipendente).setUfficio("Personale");
...
```

L'espressione logica

oggetto **instanceof** *Classe*

viene valutata **true** se e solo se *oggetto* è un'istanza di *Classe*: è richiesto che *Classe* sia staticamente il nome del tipo; non è possibile utilizzare riferimenti variabili.

Una buona pratica di programmazione OO utilizza quanto più possibile il polimorfismo limitando allo stretto indispensabile il ricorso all'operatore **instanceof**.

6.1 Un esempio di gerarchia di classi

Concludiamo il paragrafo presentando un esempio di gerarchia di classi relativa alle merci in vendita in un supermercato: la classe astratta *Merce* è la classe principale della gerarchia che fattorizza le caratteristiche comuni a tutti i prodotti in vendita: codice e prezzo unitario.

La classe *Merce* definisce un metodo astratto per il calcolo dell'importo relativo a ogni prodotto acquistato in base a eventuali sconti o al peso della merce acquistata. La merce è suddivisa in due categorie modellate da due classi concrete che derivano da *Merce*: *Abbigliamento* e *Alimentari*. Quest'ultima è a sua volta una classe astratta da cui derivano due classi concrete, *Freschi* e *Conservati*, rispettivamente relative ai prodotti di un tipo o dell'altro.

Per tutti gli alimentari si tiene conto del contenuto calorico unitario, mentre il prezzo unitario è inteso per chilogrammo di prodotto e l'importo da pagare viene determinato moltiplicando l'importo unitario per il peso del prodotto acquistato; inoltre, per gli alimentari freschi non confezionati, è previsto il costo aggiuntivo del sacchetto necessario per il trasporto. Per gli articoli di abbigliamento sono previste due distinte percentuali di sconto a seconda che si tratti di articoli da uomo o da donna.

La classe *Carrello* viene infine utilizzata come contenitore della merce da pagare alla cassa: la cassa prevede la visualizzazione dei singoli prodotti contenuti nel carrello, che devono quindi essere esportati, la visualizzazione dell'importo totale da pagare e delle calorie complessive relative agli alimentari acquistati. Il diagramma UML delle classi di FIGURA 9 descrive la soluzione adottata.

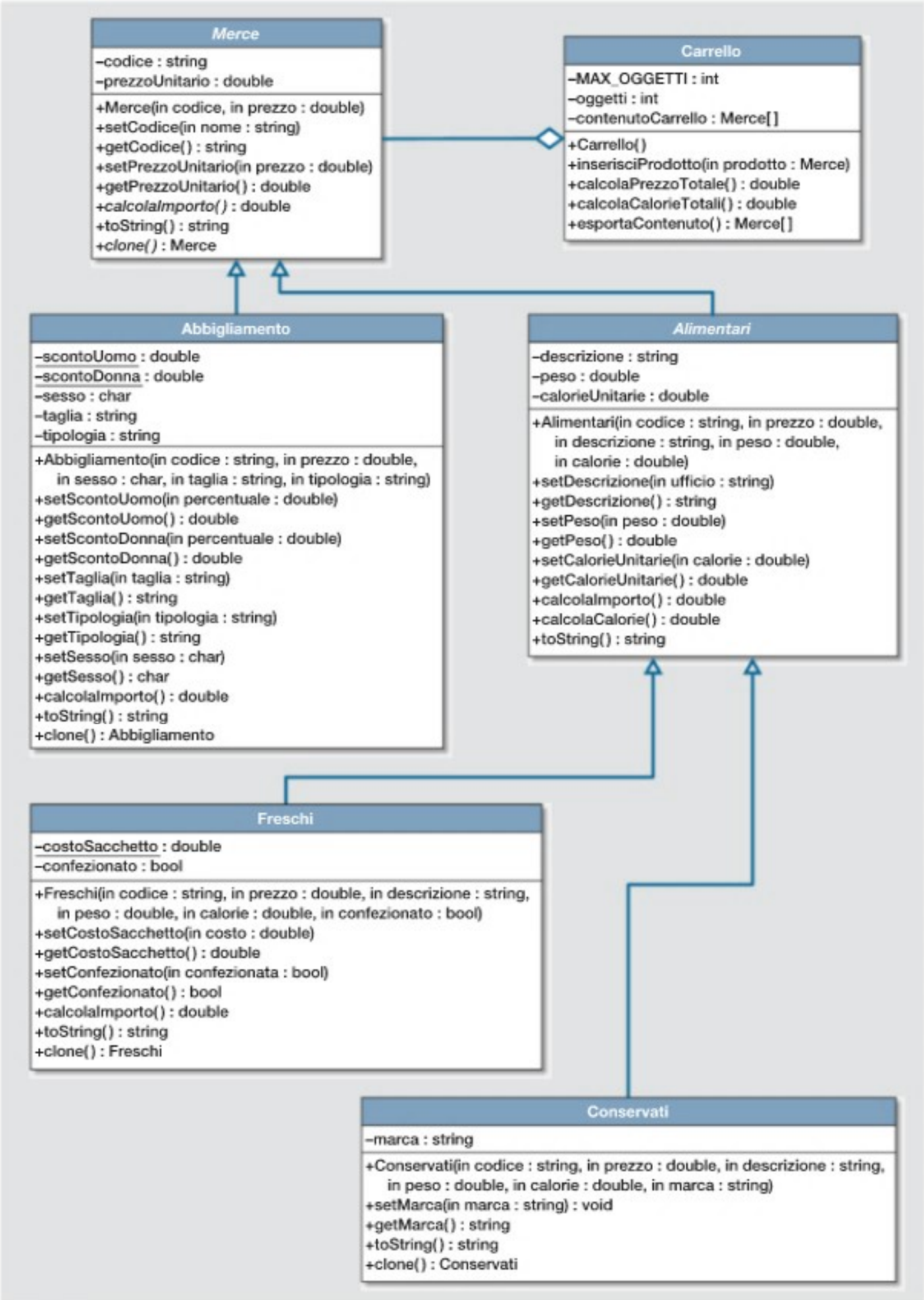

Merce
-codice : string
-prezzoUnitario : double
+Merce(in codice, in prezzo : double)
+setCodice(in nome : string)
+getCodice() : string
+setPrezzoUnitario(in prezzo : double)
+getPrezzoUnitario() : double
+calcolaImporto() : double
+toString() : string
+clone() : Merce
Carrello
-MAX_OGGETTI : int
-oggetti : int
-contenutoCarrello : Merce[]
+Carrello()
+inserisciProdotto(in prodotto : Merce)
+calcolaPrezzoTotale() : double
+calcolaCalorieTotali() : double
+esportaContenuto() : Merce[]
Abbigliamento
-scontoUomo : double
-scontoDonna : double
-sesso : char
-taglia : string
-tipologia : string
+Abbigliamento(in codice : string, in prezzo : double, in sesso : char, in taglia : string, in tipologia : string)
+setScontoUomo(in percentuale : double)
+getScontoUomo() : double
+setScontoDonna(in percentuale : double)
+getScontoDonna() : double
+setTaglia(in taglia : string)
+getTaglia() : string
+setTipologia(in tipologia : string)
+getTipologia() : string
+setSesso(in sesso : char)
+getSesso() : char
+calcolaImporto() : double
+toString() : string
+clone() : Abbigliamento
Alimentari
-descrizione : string
-peso : double
-calorieUnitarie : double
+Alimentari(in codice : string, in prezzo : double, in descrizione : string, in peso : double, in calorie : double)
+setDescrizione(in ufficio : string)
+getDescrizione() : string
+setPeso(in peso : double)
+getPeso() : double
+setCalorieUnitarie(in calorie : double)
+getCalorieUnitarie() : double
+calcolaImporto() : double
+calcolaCalorie() : double
+toString() : string
Freschi
-costoSacchetto : double
-confezionato : bool
+Freschi(in codice : string, in prezzo : double, in descrizione : string, in peso : double, in calorie : double, in confezionato : bool)
+setCostoSacchetto(in costo : double)
+getCostoSacchetto() : double
+setConfezionato(in confezionata : bool)
+getConfezionato() : bool
+calcolaImporto() : double
+toString() : string
+clone() : Freschi
Conservati
-marca : string
+Conservati(in codice : string, in prezzo : double, in descrizione : string, in peso : double, in calorie : double, in marca : string)
+setMarca(in marca : string) : void
+getMarca() : string
+toString() : string
+clone() : Conservati

FIGURA 9

Il codice Java che segue implementa il diagramma UML:

```
abstract public class Merce {
  private String codice;
  protected double prezzoUnitario;

  public Merce(String codice, double prezzo) {
    this.codice = codice;
    this.prezzoUnitario = prezzo;
  }

  public double getPrezzoUnitario() {
    return prezzoUnitario;
  }

  public String getCodice() {
    return codice;
  }

  public void setPrezzoUnitario(double prezzo) {
    this.prezzoUnitario = prezzo;
  }

  public void setCodice(String codice) {
    this.codice=codice;
  }

  abstract public double calcolaImporto();
  abstract public Merce clone();

  public String toString() {
    return "Codice: "+codice+" Prezzo unitario: "+prezzoUnitario;
  }
}

public class Abbigliamento extends Merce {
  private static double scontoDonna = 10.0;
  private static double scontoUomo = 5.0;
  private char sesso;
  private String taglia;
  private String tipologia;

  public Abbigliamento(String codice, double prezzo, char sesso,
                       String taglia, String tipologia) {
    super(codice, prezzo);
    this.sesso = sesso;
    this.taglia = taglia;
    this.tipologia = tipologia;
  }

  static public double getScontoDonna() {
    return scontoDonna;
  }
```

```
static public void setScontoDonna(double percentuale) {
  scontoDonna = percentuale;
}

static public double getScontoUomo() {
  return scontoUomo;
}

static public void setScontoUomo(double percentuale) {
  scontoUomo = percentuale;
}

public void setSesso(char sesso) {
  this.sesso = sesso;
}

public char getSesso() {
  return sesso;
}

public void setTaglia(String taglia) {
  this.taglia = taglia;
}

public String getTaglia() {
  return taglia;
}

public void setTipologia(String tipologia) {
  this.tipologia = tipologia;
}

public String getTipologia() {
  return tipologia;
}

public double calcolaImporto() {
  if (sesso == 'F')
    return prezzoUnitario-prezzoUnitario*scontoDonna/100.0;
  if (sesso == 'M')
    return prezzoUnitario-prezzoUnitario*scontoUomo/100.0;

  return prezzoUnitario;
}

public String toString() {
  return super.toString()+" Tipologia: "+tipologia+" Sesso: "+sesso+
         " Taglia: "+taglia+" Prezzo: "+calcolaImporto();
}

public Abbigliamento clone() {
  Abbigliamento prodotto = new Abbigliamento(super.getCodice(),
                                             super.prezzoUnitario,
                                             this.sesso, this.taglia,
                                             this.tipologia );
```

```
    return prodotto;
  }
}

abstract public class Alimentari extends Merce {
  private String descrizione;
  protected double peso;
  private double calorieUnitarie;

  public Alimentari(String codice, double prezzo, String descrizione,
                    double peso, double calorie) {
    super(codice, prezzo);
    this.descrizione = descrizione;
    this.peso = peso;
    this.calorieUnitarie = calorie;
  }

  public void setDescrizione(String descrizione) {
    this.descrizione = descrizione;
  }

  public void setPeso(double peso) {
    this.peso = peso;
  }

  public void setCalorieUnitarie(double calorie) {
    this.calorieUnitarie = calorie;
  }

  public String getDescrizione() {
    return descrizione;
  }

  public double getPeso() {
    return peso;
  }

  public double getCalorieUnitarie() {
    return calorieUnitarie;
  }

  public double calcolaCalorie() {
    return peso*calorieUnitarie;
  }

  public double calcolaImporto() {
    return super.prezzoUnitario*peso;
  }

  public String toString() {
    return super.toString()+" Descrizione: "+descrizione+" Peso: "+peso
           +" Importo: "+calcolaImporto()+" Calorie: "+
           calcolaCalorie();
  }
}
```

```java
public class Freschi extends Alimentari {
  static private double costoSacchetto = 0.05;
  private boolean confezionato;

  public Freschi(String codice, double prezzo, String descrizione,
                 double peso, double calorie, boolean confezionato) {
    super(codice, prezzo, descrizione, peso, calorie);
    this.confezionato = confezionato;
  }

  static public double getCostoSacchetto() {
    return costoSacchetto;
  }

  static public void setCostoSacchetto(double costo) {
    costoSacchetto = costo;
  }

  public void setConfezionato(boolean confezionato) {
    this.confezionato = confezionato;
  }

  public boolean getConfezionato() {
    return confezionato;
  }

  public String toString() {
    String tipoConfezione;

    if (!confezionato)
      tipoConfezione = "sfusa";
    else
      tipoConfezione = "preconfezionata";
    return super.toString()+" Confezione: "+tipoConfezione;
  }

  public double calcolaImporto() {
    double importo = super.calcolaImporto();

    if (!confezionato)
      importo += costoSacchetto;
    return importo;
  }

  public Freschi clone() {
    Freschi prodotto = new Freschi(super.getCodice(),super.prezzoUnitario,
                                   super.getDescrizione(),super.getPeso(),
                                   super.getCalorieUnitarie(),
                                   this.confezionato);
    return prodotto;
  }
}
```

```
public class Conservati extends Alimentari {
  private String marca;

  public Conservati (String codice, double prezzo, String descrizione,
                     double peso, double calorie, String marca) {
    super(codice, prezzo, descrizione, peso, calorie);
    this.marca = marca;
  }

  public void setMarca(String marca) {
    this.marca = marca;
  }

  public String getMarca() {
    return marca;
  }

  public String toString() {
    return super.toString()+" Marca: "+marca;
  }

  public Conservati clone() {
    Conservati prodotto = new Conservati(super.getCodice(),
                                         super.prezzoUnitario,
                                         super.getDescrizione(),
                                         super.getPeso(),
                                         super.getCalorieUnitarie(),
                                         this.marca);
    return prodotto;
  }
}

public class Carrello {
  private final int MAX_OGGETTI = 100;
  private int oggetti;
  private Merce[] contenutoCarrello;

  public Carrello() {
    contenutoCarrello = new Merce[MAX_OGGETTI];
    oggetti=0;
  }

  public void inserisciProdotto(Merce prodotto) {
    if (oggetti < MAX_OGGETTI) {
      contenutoCarrello[oggetti] = prodotto.clone();
      oggetti++;
    }
  }

public double calcolaPrezzoTotale() {
    double prezzoTotale=0;

    for (int i=0; i<oggetti; i++)
      prezzoTotale += contenutoCarrello[i].calcolaImporto();
```

```
    return prezzoTotale;
  }

  public double calcolaCalorieTotali() {
    double calorieTotali=0;

    for (int i=0; i<oggetti; i++)
      if (contenutoCarrello[i] instanceof Alimentari)
        calorieTotali +=
        ((Alimentari)contenutoCarrello[i]).calcolaCalorie();

    return calorieTotali;
  }

  public Merce[] esportaContenuto() {
    Merce prodotti[] = new Merce[oggetti];

    for (int i=0; i<oggetti; i++)
      prodotti[i] = contenutoCarrello[i].clone();

    return prodotti;
  }
}
```

Nella classe *Carrello* è utilizzato un vettore di oggetti di tipo *Merce* per contenere i prodotti della spesa inseriti nel carrello (fino a un numero massimo impostato con la costante *MAX_OGGETTI*). Essendo i prodotti disponibili di tipo eterogeneo (abbigliamento, alimentari freschi e alimentari conservati), il polimorfismo consente di assegnare agli elementi del vettore qualsiasi oggetto istanza di una classe derivata da *Merce*. La selezione del codice corretto da eseguire in risposta all'invocazione di uno dei metodi polimorfici – *calcolaImporto*, *toString* e *clone* – avviene applicando il *binding* dinamico. L'operatore `instanceof` viene utilizzato per verificare, nel metodo che calcola le calorie alimentari totalizzate nel carrello della spesa, che un oggetto sia effettivamente di tipo *Alimentari* prima di invocare il metodo *calcolaCalorie*. Il metodo polimorfico *clone* è stato implementato per ciascuna delle classi della gerarchia ed è invocato dal metodo *esportaContenuto*.

Il metodo *main* che segue dimostra le funzionalità della classe *Carrello*, in particolare la visualizzazione dei prodotti presenti nel carrello invoca in modo polimorfico il metodo *toString*:

```
public static void main(String[] args) {
  Carrello carrello = new Carrello();

  Abbigliamento abbigliamento1 = new Abbigliamento("PU01", 50.00, 'M',"XL",
                                                   "Pantaloni");
  Abbigliamento abbigliamento2 = new Abbigliamento("CD22", 35.00,'F',"XS",
                                                   "Camicia");
  Abbigliamento abbigliamento3 = new Abbigliamento("GD46", 29.00, 'F',"S",
                                                   "Giacca");
  Freschi freschi1 = new Freschi("OR044", 1.50, "Peperoni", 1.4, 170, true);   ▶
```

```
    Freschi freschi2 = new Freschi("FR012", 1.90, "Mele", 0.5, 730, false);
    Conservati conservati = new Conservati("BS003", 2.00, "Biscotti",
                                           0.37, 1200, "Buoni e belli");

    carrello.inserisciProdotto(abbigliamento1);
    carrello.inserisciProdotto(abbigliamento2);
    carrello.inserisciProdotto(abbigliamento3);
    carrello.inserisciProdotto(freschi1);
    carrello.inserisciProdotto(freschi2);
    carrello.inserisciProdotto(conservati);

    Merce prodotti[] = carrello.esportaContenuto();
    for (int i=0; i<prodotti.length; i++)
      System.out.println(prodotti[i]);

    System.out.println("Spesa totale: "+ carrello.calcolaPrezzoTotale());
    System.out.println("Calorie totali: " + carrello.calcolaCalorieTotali());
  }
```

L'output prodotto è il seguente:

```
Codice: PU01 Prezzo unitario: 50.0 Tipologia: Pantaloni Sesso: M
Taglia: XL Prezzo: 47.5
Codice: CD22 Prezzo unitario: 35.0 Tipologia: Camicia Sesso: F
Taglia: XS Prezzo: 31.5
Codice: GD46 Prezzo unitario: 29.0 Tipologia: Giacca Sesso: F
Taglia: S Prezzo: 26.1
Codice: OR044 Prezzo unitario: 1.5 Descrizione: Peperoni Peso: 1.4
Importo: 2.0999999999999996 Calorie: 237.99999999999997 Confezione:
preconfezionata
Codice: FR012 Prezzo unitario: 1.9 Descrizione: Mele Peso: 0.5
Importo: 1.0 Calorie: 365.0 Confezione: sfusa
Codice: BS003 Prezzo unitario: 2.0 Descrizione: Biscotti Peso: 0.37
Importo: 0.74 Calorie: 444.0 Marca: Buoni e belli
Spesa totale: 108.93999999999998
Calorie totali: 1047.0
```

7 Gerarchie di eccezioni e loro gestione

Le eccezioni nel linguaggio Java sono classi che derivano dalla classe *Exception*: è quindi possibile ricorrere all'ereditarietà per costruire gerarchie di eccezioni e sfruttare il polimorfismo per una loro più agevole gestione.

ESEMPIO

Nel capitolo A3 è stata introdotta la classe *TextFile* per la scrittura/lettura sequenziale di file testuali: i metodi della classe generano eccezioni di classe *FileException* fornendo come parametro al costruttore una stringa di descrizione dell'errore che può essere in seguito recuperata mediante un metodo specifico della classe-eccezione. Un modo più razionale consiste nel definire la seguente gerarchia di eccezioni

```
abstract public class FileException extends Exception {
}
```

```
public class ReadOnlyFile extends FileException {
  public String toString() {
    return "Read-only file!";
  }
}

public class WriteOnlyFile extends FileException {
  public String toString() {
    return "Write-only file!";
  }
}

public class EndOfFile extends FileException {
  public String toString() {
    return "End of file!";
  }
}
```

che può essere gestita in modo polimorfico indicando nella clausola **throws** dei metodi la sola classe astratta *FileException* da cui derivano le eccezioni concretamente generate:

```
// classe per la gestione sequenziale di un file di testo
public class TextFile {
  private char mode; // R=read, W=write
  private BufferedReader reader;
  private BufferedWriter writer;

  // costruisce un oggetto di tipo BufferedReader/BufferedWriter
  // sopra il file specificato dal nome indicato
  public TextFile(String filename, char mode) throws IOException
  {
    this.mode = 'R';
    if (mode == 'W' || mode == 'w') {
      this.mode = 'W';
      writer = new BufferedWriter(new FileWriter(filename));
    }
    else {
      reader = new BufferedReader(new FileReader(filename));
    }
  }

  // scrive una riga di testo in un file aperto in scrittura
  public void toFile(String line) throws FileException, IOException
  {
    if (this.mode == 'R')
      throw new ReadOnlyFile();
    writer.write(line);
    writer.newLine();
  }

  // legge una riga di testo da un file aperto in lettura
  public String fromFile() throws FileException, IOException
  {
    String tmp;

    if (this.mode == 'W')
      throw new WriteOnlyFile();
```

```
    tmp = reader.readLine();
    if (tmp == null)
      throw new EndOfFile();
    return tmp;
  }

  // chiude il file aperto dal costruttore
  public void closeFile() throws IOException
  {
    if (this.mode == 'W')
      writer.close();
    else
      reader.close();
  }
}
```

Com'è illustrato dal seguente metodo *main* di test della classe *TextFile*, è in questo caso sufficiente una sola clausola `catch` del blocco `try` per intercettare tutte le eccezioni che ereditano dalla classe *FileException*; il polimorfismo ne consente la corretta gestione (in questo caso limitata alla visualizzazione della tipologia di errore):

```
public static void main(String args[]) throws IOException
{
  TextFile out = new TextFile("file.txt",'W');
  try {
    out.toFile("Riga 1");
    out.toFile("Riga 2");
    out.toFile("Riga 3");
    String tmp = out.fromFile();
  }
  catch (FileException exception) {
      System.out.println(exception);
  }
  out.closeFile();

  TextFile in = new TextFile("file.txt",'R');
  try {
    while (true) {
      String line = in.fromFile();
      System.out.println(line);
    }
  }
  catch (FileException exception) {
    System.out.println(exception);
  }
  out.closeFile();
}
```

Come caso limite si ha che una clausola `catch` con parametro di tipo *Exception* intercetta tutte le eccezioni generate dal codice inserito nel blocco `try`.

Un aspetto non secondario del polimorfismo del linguaggio Java è dato dalla regola relativa alle eccezioni dei metodi ridefiniti in una classe derivata: un metodo che ridefinisce un metodo che dichiara di generare eccezioni non può sollevare tipi di eccezione diversi o aggiuntivi rispetto a quelli del metodo originale. Questa regola assicura che tutte le eccezioni generate siano sempre gestite.

Se, per esempio, il metodo originale genera eccezioni di classe *Eccezione*, il corrispondente metodo ridefinito non può sollevare eccezioni di una superclasse di *Eccezione*, o di una classe diversa da *Eccezione*, ma può:

- generare eccezioni di classe *Eccezione*;
- generare eccezioni di una sottoclasse di *Eccezione*;
- non generare eccezioni.

Se non esistesse la limitazione esposta si avrebbero situazioni incoerenti per quanto riguarda la gestione delle eccezioni, come esemplificato nel codice che segue:

```
public class Alfa {
  public void esegui() throws Eccezione {
    ...
    ...
    ...
  }
}

public class Beta {
  public void metodo(Alfa a) {
    try {
      a.esegui();
    }
    catch (Eccezione e) {
      ...
    }
  }
}
```

Il polimorfismo consente di invocare il metodo *metodo* della classe *Beta* fornendo come parametro un argomento che è una sottoclasse di *Alfa* e, come conseguenza dell'*overriding* e del *binding* dinamico, non è certo che il metodo *esegui* invocato dal codice metodo sia quello definito nella classe *Alfa*: potrebbe essere stato ridefinito nella sottoclasse, ma la regola enunciata impedisce che esso sollevi eccezioni potenzialmente non previste. Tale situazione, infatti, potrebbe verificarsi solo se il metodo ridefinito sollevasse eccezioni che non sono istanze della classe *Eccezione*, o di una sua sottoclasse.

Un diverso esempio di uso improprio, e quindi illegale, delle eccezioni del linguaggio Java è documentato dal seguente codice:

```
public class Alfa {
  public void esegui() {
    ...
    ...
    ...
  }
}

public class Beta extends Alfa {
  public void esegui() throws Eccezione {
```

```
    ...
    ...
  }
}
```

In questo caso il polimorfismo permetterebbe il seguente frammento di codice:

```
...
Alfa a = new Beta();
...
a.esegui();
...
```

Ma l'invocazione del metodo *esegui* può sollevare un'eccezione di tipo *Eccezione* non dichiarata – e quindi non intercettabile – nella classe *Alfa*.

Le eccezioni generate da parte delle sottoclassi possono essere solo un sottoinsieme delle eccezioni generate dalla superclasse.

I CONCETTI CHIAVE

EREDITARIETÀ. È un meccanismo di astrazione finalizzato alla creazione di gerarchie di classi: nel linguaggio Java è realizzato mediante la parola chiave **extends** utilizzata come nella seguente dichiarazione

```
public class B extends A {...}
```

dove *A* è detta superclasse e *B* classe derivata, o sottoclasse; la classe *A* rappresenta una generalizzazione della classe *B*, mentre questa è una specializzazione della classe *A*. La classe *B* eredita attributi e metodi della classe *A*, può introdurne di nuovi e ridefinirne alcuni. Tutte le classi definite in linguaggio Java derivano implicitamente dalla classe «radice» *Object*. L'ereditarietà viene spesso usata per fattorizzare caratteristiche (attributi e metodi) comuni a più classi.

OVERRIDING. Consiste nella ridefinizione in una classe derivata di metodi già definiti nella superclasse: per gli oggetti istanza di una classe derivata un metodo sovrascritto nasconde la definizione corrispondente data nella superclasse.

HASHCODE. La classe *Object* definisce i metodi *toString* ed *equals*, che sono ereditati da qualsiasi classe definita in un programma Java: un uso coerente di questi metodi ne richiede la ridefinizione *ad hoc* nel contesto di una specifica classe. Essi infatti operano sullo *hashCode* di un oggetto che l'ambiente di esecuzione Java calcola a partire dall'indirizzo di memorizzazione nello *heap* piuttosto che sul contenuto informativo. In particolare il metodo *equals* definito nella classe *Object* considera sempre diversi due oggetti distinti che hanno necessariamente indirizzi di memoria diversi.

super. Questa parola chiave del linguaggio Java permette di far riferimento ai metodi e agli attributi definiti nella superclasse nel codice dei metodi di una classe derivata. Utilizzando la stessa parola chiave è possibile invocare il costruttore della superclasse nel codice del costruttore di una classe derivata. L'uso corretto della parola chiave **super** garantisce il rispetto del principio di incapsulamento e rende indipendente il codice di una classe derivata da eventuali modifiche del codice della superclasse.

UP-CASTING* E *DOWN-CASTING. È consentito far riferire un oggetto istanza di una certa classe da una variabile avente come tipo una qualsiasi superclasse, ma non viceversa. Il principio da seguire è quello per cui un oggetto riferito da una variabile di diverso tipo debba avere tutti i requisiti affinché l'invocazione dei metodi della classe che definisce il tipo della variabile abbia successo. Nel linguaggio di programmazione Java è possibile operare il *casting* dei tipi degli oggetti in modo che un riferimento a un oggetto possa essere trasformato in un riferimento di diverso tipo. Le operazioni di *casting* di un riferimento possono essere utilizzate per trasformare il riferimento a un oggetto in un riferimento avente come tipo una superclasse – si tratta in questo caso di *up-casting* – oppure in un riferimento a una sottoclasse – si tratta allora di un *down-casting* (nel caso di *up-casting* non è necessario il ricorso esplicito all'operatore di *cast* perché il *casting* è implicito). Ogni oggetto può essere assegnato a una variabile di classe *Object*. Le regole di compilazione non permettono di operare in tutti quei casi in cui il *casting* non è possibile (quando, per esempio, le classi delle variabili non hanno tra loro alcuna relazione gerarchica di ereditarietà), ma in ogni caso in fase di esecuzione del programma (*run-time*) si può verificare un'eccezione di tipo *ClassCastException* se l'oggetto del *casting* non è compatibile con il tipo della variabile a cui viene assegnato.

METODO *CLONE*. È un metodo definito dalla classe *Object* che restituisce una copia dell'oggetto su cui viene invocato.

DEEP COPY* E *SHALLOW COPY. Esistono due distinti criteri per considerare due oggetti uguali:

1) gli attributi corrispondenti assumono lo stesso valore se sono di tipo primitivo, o riferiscono lo stesso oggetto se sono riferimenti;
2) gli attributi corrispondenti assumono lo stesso valore se sono di tipo primitivo, o riferiscono oggetti che sono ricorsivamente uguali se sono riferimenti.

Dato che un'operazione di copia ha come scopo la creazione di un nuovo oggetto «uguale» a un oggetto preesistente, esistono di conseguenza due diverse tipologie di copia:

1) *shallow copy* (copia superficiale), in cui viene creato un nuovo oggetto uguale all'originale secondo il criterio di uguaglianza 1; gli attributi del nuovo oggetto creato hanno lo stesso valore dell'oggetto originale, sia che si tratti di tipi primitivi sia di riferimenti;
2) *deep copy* (copia profonda), in cui viene creato un nuovo oggetto uguale all'originale secondo il criterio di uguaglianza 2; gli attributi del nuovo oggetto, se non sono di tipo primitivo, nel qual caso assumono lo stesso valore, sono ricorsivamente copie profonde degli attributi corrispondenti dell'oggetto originale.

Premesso che i due tipi di copia coincidono nel caso di oggetti in cui tutti gli attributi sono di tipo primitivo, se si desidera disporre di un meccanismo di copia in cui l'oggetto creato risulti completamente indipendente in tutte le sue componenti dall'oggetto originale (cioè senza riferimenti comuni), dato che il metodo *clone* ereditato dalla classe *Object* implementa una *shallow copy*, è necessario realizzare la *deep copy* ridefinendo il metodo *clone* stesso.

CLASSI ASTRATTE. In una gerarchia di classi è conveniente fattorizzare attributi e metodi comuni a più classi al livello più alto della gerarchia, in modo che possano essere ereditati dal maggior numero possibile di classi. In questo contesto le classi astratte consentono di usare questo meccanismo di fattorizzazione degli attributi comuni anche nel caso in cui la superclasse risultante non sia «ben definita». Nel linguaggio Java una classe astratta viene definita premettendo al suo nome il qualificatore `abstract` e non può essere istanziata anche se può avere costruttori. Una classe astratta può avere uno o più «metodi astratti», cioè metodi privi del corpo e qualificati con la parola chiave `abstract` la cui implementazione è vincolante per le classi derivate; la dichiarazione di uno o più metodi astratti rende implicitamente una classe astratta. Se una classe concreta deriva da una classe astratta *deve* fornire un'implementazione per tutti i metodi astratti ereditati.

INTERFACCE. Nel linguaggio di programmazione Java la parola chiave `interface` consente di dichiarare una *interfaccia* che è analoga a una classe, ma a differenza di questa costituisce una pura specifica di invocazione e come tale si limita a dichiarare i metodi, definendone la firma, senza implementarli (tutti i metodi di un'interfaccia sono implicitamente astratti). Un'interfaccia viene *implementata* dalle classi mediante l'uso della parola chiave `implements`. Implementare un'interfaccia impone formalmente a una classe di definire i metodi che l'interfaccia stessa dichiara: in questo senso le interfacce costituiscono una specie di «contratto» per le classi che le implementano e il rispetto di tale contratto viene verificato dal compilatore che genera un errore nel caso in cui una classe che dichiara di implementare una determinata interfaccia non ne definisca tutti i metodi dichiarati. Dal momento che in Java una classe può implementare più di un'interfaccia, questo meccanismo permette di simulare l'ereditarietà multipla di cui il linguaggio non dispone.

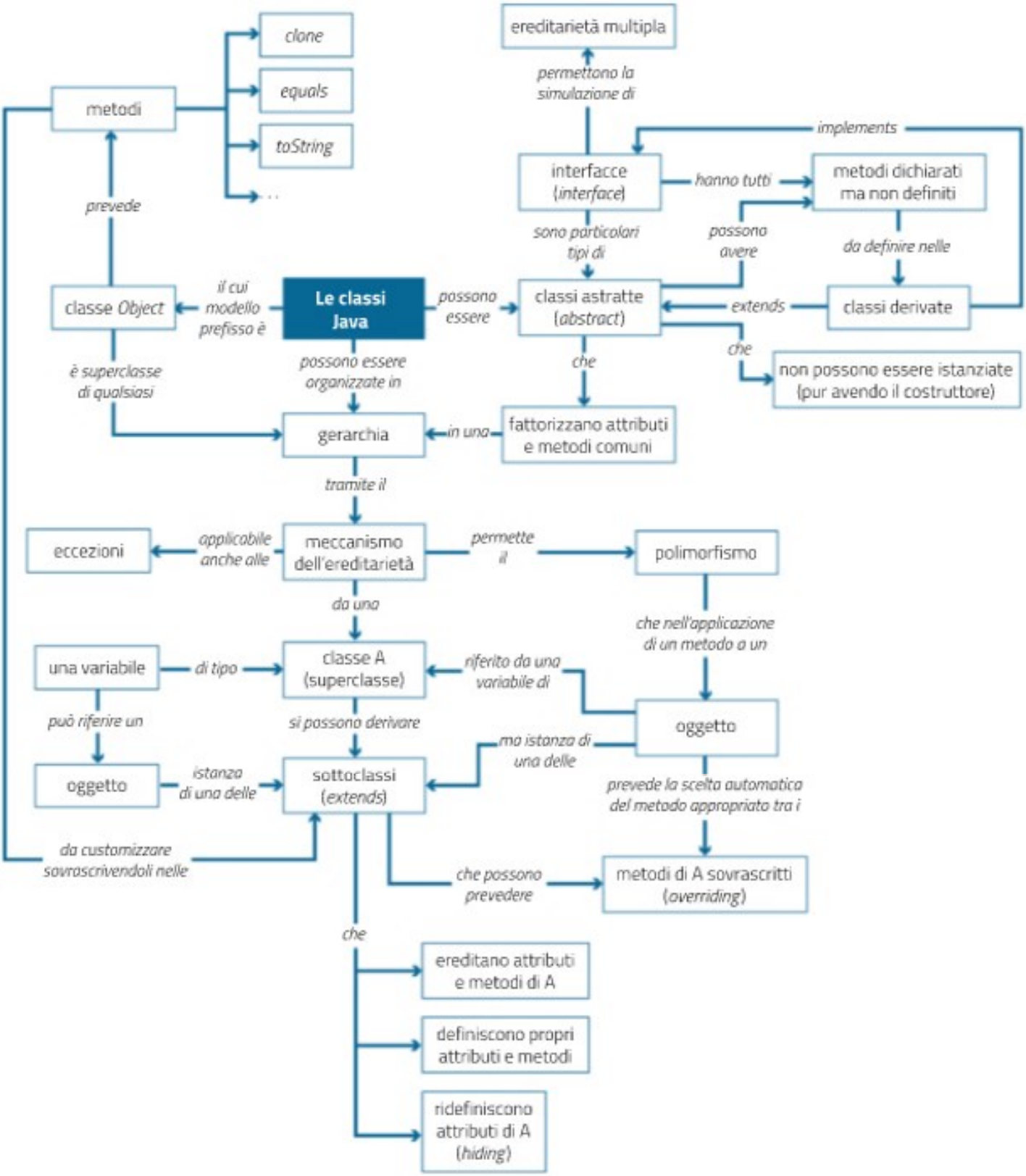
POLIMORFISMO. L'operazione di estensione di una classe permette di derivare da essa una o più sottoclassi: è sempre possibile utilizzare un oggetto istanza di una delle sottoclassi derivate in un contesto in cui sia richiesto un oggetto istanza della classe originale. Questa caratteristica dei linguaggi di programmazione OO in generale e del linguaggio Java in particolare viene detta polimorfismo, intendendo con questo termine la capacità di un oggetto di assumere «forme» molteplici: istanza della propria classe o istanza di ogni superclasse di essa. Il polimorfismo degli oggetti risulta particolarmente utile quando, in una gerarchia di classi derivate, viene ridefinito un metodo e la specifica versione del metodo da eseguire viene scelta automaticamente sulla base del tipo di oggetto effettivamente assegnato a un riferimento in fase di esecuzione, piuttosto che al momento della compilazione.

***BINDING* STATICO E *BINDING* DINAMICO**. Il meccanismo che determina quale metodo deve essere invocato in base alla classe di appartenenza dell'oggetto è definito *binding* (letteralmente «legame», «collegamento»). Esistono due tipi fondamentali di *binding*: *binding* statico (*early binding*), in cui il metodo da invocare viene determinato in fase di compilazione del codice (*compile-time binding*), e *binding* dinamico (*late binding*), in cui il metodo da invocare viene determinato durante l'esecuzione del codice (*run-time binding*). Il linguaggio Java, come la maggior parte dei linguaggi di programmazione a oggetti, adotta il *binding* dinamico a *run-time* quando il *binding* statico a *compile-time* risulta impossibile, come nel caso dei metodi polimorfici. Il polimorfismo consente di sviluppare codice a oggetti *estensibile*: l'aggiunta di nuove classi a una gerarchia preesistente non influenza il codice che gestisce oggetti istanza delle classi della gerarchia stessa in modo polimorfico.

RTTI (*Run-Time Type Identification*). L'esigenza di un supporto per l'identificazione del tipo effettivo di un oggetto durante l'esecuzione di un programma in linguaggio Java nasce dall'impossibilità di invocare direttamente il metodo di una sottoclasse su un riferimento di una superclasse, cosa che può risultare utile o anche necessaria. Per applicare propriamente il *casting* necessario in questa situazione è indispensabile conoscere il tipo effettivo di cui l'oggetto riferito è istanza: l'operatore Java `instanceof` permette di verificare la classe di appartenenza di un oggetto. L'espressione logica `oggetto instanceof Classe` viene valutata `true` se e solo se *oggetto* è un'istanza di *Classe*: è richiesto che *Classe* sia staticamente il nome del tipo; non è possibile utilizzare riferimenti variabili.

ECCEZIONI E POLIMORFISMO. Un aspetto non secondario del polimorfismo del linguaggio Java è dato dalla regola relativa alle eccezioni dei metodi ridefiniti in una classe derivata: un metodo che ridefinisce un metodo che dichiara di generare eccezioni non può sollevare tipi di eccezione diversi o aggiuntivi rispetto a quelli del metodo originale. Questa regola assicura che tutte le eccezioni generate siano sempre correttamente gestite.

RIPASSA CON LA MAPPA



SPIEGA IN UNA PAGINA

1 **Descrivere le caratteristiche base del meccanismo relativo al polimorfismo.**

2 **Descrivere quali possono essere le necessità che possono indurre alla definizione di una classe astratta.**

QUESITI A RISPOSTA APERTA

3 **Come si realizza l'ereditarietà nel linguaggio Java?**

4 **Che cos'è il metodo *clone*?**

5 **Che cosa sono le interfacce nel linguaggio Java?**

6 **Come funziona il metodo della *deep copy*?**

7 **Che cos'è l'*overriding*?**

TEST A RISPOSTA MULTIPLA

8 **L'ereditarietà è un meccanismo di astrazione finalizzato:**

A alla definizione di interfacce.
B all'implementazione dell'*information hiding*.
C alla costruzione di gerarchie classi.
D alla gestione degli attributi complessi.

9 **Rispetto all'ereditarietà, il linguaggio Java:**

A non ne permette l'implementazione.
B permette l'implementazione dell'ereditarietà singola.
C permette l'implementazione dell'ereditarietà multipla.
D permette l'implementazione dell'ereditarietà multipla solo per le classi astratte.

10 **In un contesto di ereditarietà tra due classi:**

A la classe derivata rappresenta una generalizzazione della superclasse.
B la superclasse rappresenta una specializzazione della classe derivata.
C la classe derivata e la superclasse sono paritetiche.
D la classe derivata rappresenta una specializzazione della superclasse.

11 **In un contesto di ereditarietà tra due classi:**

A la classe derivata eredita attributi e metodi della superclasse.
B la superclasse eredita attributi e metodi della classe derivata.
C la classe derivata eredita solo i metodi della superclasse.
D la classe derivata eredita solo gli attributi della superclasse.

12 **Il meccanismo di *overriding* permette di:**

A ridefinire nella classe derivata metodi definiti nella superclasse.
B ridefinire nella superclasse metodi definiti nella classe derivata.
C definire nell'ambito di una stessa classe più metodi con lo stesso nome.
D ridefinire in una classe i metodi definiti dalla classe *Object*.

13 **La classe *Object* Java è:**

A la classe derivata di una qualsiasi altra classe.
B una classe di sistema che non può essere derivata.
C è la classe radice di una qualsiasi gerarchia di classi.
D un'interfaccia.

14 **I metodi *toString* ed *equals*:**

A sono metodi della classe *Object* che per poter essere adeguatamente utilizzati devono essere sovrascritti nelle classi in cui si è interessati a utilizzarli.
B sono metodi della classe *Object* che non possono essere sovrascritti in altre classi.
C non sono metodi della classe *Object*.
D sono metodi della classe *Class*.

15 **Il metodo *equals* serve a confrontare:**

A due oggetti per verificare se i loro attributi hanno lo stesso valore.
B due oggetti per verificare se sono o meno lo stesso oggetto.
C due classi per verificare se sono identiche.
D gli attributi corrispondenti di due oggetti distinti.

16 **Il metodo *toString* è un metodo della classe *Object* che serve a:**

A convertire una classe in una stringa.

B ottenere una stringa che descrive l'elenco delle classi di una gerarchia.

C ottenere i valori degli attributi di un oggetto in formato stringa.

D ottenere l'elenco degli identificatori dei metodi di una classe.

17 **Con «fattorizzazione delle caratteristiche» si intende la possibilità di:**

A raggruppare, utilizzando l'ereditarietà, caratteristiche comuni a più classi evitando così inutili ridondanze.

B duplicare, utilizzando l'ereditarietà, caratteristiche comuni a più classi.

C ridondare le caratteristiche comuni alle varie classi di una gerarchia.

D modificare le caratteristiche comuni a più classi.

18 **La parola chiave `super` del linguaggio Java può essere usata per:**

A invocare in una sottoclasse un metodo della superclasse che è stato sovrascritto.

B invocare da una superclasse un metodo sovrascritto in una sottoclasse.

C invocare in una classe derivata il costruttore di una qualsiasi delle superclassi.

D riferire in una classe derivata gli attributi ereditati.

19 **È possibile far riferire un oggetto di una certa classe:**

A esclusivamente da una variabile avente come tipo la classe di cui questa è estensione diretta.

B da una variabile avente come tipo una qualsiasi superclasse.

C da una variabile avente come tipo una qualsiasi classe derivata.

D da una variabile avente come tipo una qualsiasi classe definita nello stesso *package*.

20 **In quali situazioni è sempre possibile operare il *casting* dei tipi dei riferimenti?**

A Verso la stessa classe dell'oggetto riferito.

B Verso una superclasse o una sottoclasse dell'oggetto riferito.

C Verso una classe che non appartiene alla gerarchia della classe dell'oggetto riferito.

D Verso *Object*.

21 **Il *casting* implicito si ha quando questo avviene verso:**

A una superclasse della classe dell'oggetto riferito.

B una sottoclasse della classe dell'oggetto riferito.

C la classe dell'oggetto riferito.

D una qualsiasi classe se non si specifica l'operatore di *casting*.

22 **La tecnica di applicazione del *casting* a un riferimento avente come tipo *Object* verso una sottoclasse della classe dell'oggetto riferito:**

A viene definita *up-casting*.

B viene definita *down-casting*.

C è un *casting* illegale.

D è un *casting* implicito.

23 **Il *down-casting* permette di trasformare un riferimento:**

A dalla classe radice della gerarchia verso una sottoclasse.

B di una sottoclasse verso la classe radice della gerarchia.

C da una qualsiasi classe a una qualsiasi altra classe.

D qualsiasi verso la classe *Object*.

24 **Che cos'è una *deep copy*?**

A È la copia di un oggetto che produce un nuovo oggetto i cui attributi assumono lo stesso valore (se sono di tipo primitivo), o oggetti ricorsivamente uguali (se non sono di tipo primitivo).

B È la copia di un oggetto che produce un nuovo oggetto i cui attributi assumono lo stesso valore (se sono di tipo primitivo), o lo stesso riferimento (se non sono di tipo primitivo).

C È un nome alternativo per denotare il costruttore di copia di una classe.

D È il tipo di copia realizzato dal metodo *clone* della classe *Object*.

25 **Indicare quali delle seguenti affermazioni relative a una classe astratta in linguaggio Java sono vere.**

A Non può essere estesa.

B In Java viene definita premettendo alla sua dichiarazione il qualificatore `abstract`.
C Non può essere istanziata.
D Non può avere costruttori.

26 Indicare quali delle seguenti affermazioni relative a un'interfaccia in linguaggio Java sono vere.

A Tutti i metodi di un'interfaccia sono implicitamente astratti.
B È analoga a una classe in cui tutti i metodi sono astratti.
C Una classe non può implementare più di un'interfaccia.
D Non vi sono limitazioni per la dichiarazioni degli attributi.

27 Indicare quali delle seguenti affermazioni relative al polimorfismo in linguaggio Java sono vere.

A È il meccanismo tramite il quale un oggetto ha la capacità di assumere forme molteplici: come oggetto della propria classe o come oggetto di ogni sua superclasse.
B È un meccanismo che interessa esclusivamente la dichiarazione dei metodi.
C È un meccanismo che interessa esclusivamente la dichiarazione degli attributi.
D Nel linguaggio Java il polimorfismo è basato sul *binding* statico.

28 Il polimorfismo si applica:

A a classi che non sono in relazione gerarchica.
B a classi che sono in relazione gerarchica.
C esclusivamente a classi astratte.
D solo alla classe *Object*.

29 L'acronimo RTTI significa:

A *Run-Time Type Identification.*
B *Run-Time Trouble Identification.*
C *Run-Time Type Implementation.*
D *Run-Type Time Identification.*

30 La parola chiave `instanceof` del linguaggio Java rappresenta:

A un operatore che permette di istanziare un oggetto di una specifica classe.
B un operatore che permette l'identificazione del tipo di un oggetto nella fase di esecuzione di un programma.
C un attributo della classe *Object*.
D un metodo della classe *Object*.

31 Indicare quali delle seguenti affermazioni relative alla generazione delle eccezioni da parte di metodi sovrascritti in una classe derivata sono vere.

A Possono generare a loro volta eccezioni della stessa classe di quelle generate dai rispettivi metodi della superclasse.
B Non possono generare eccezioni di sottoclassi di quelle generate dai rispettivi metodi della superclasse.
C Possono non generare eccezioni.
D Possono generare eccezioni di superclassi di quelle generate dai rispettivi metodi della superclasse.

PROBLEMI

32 Indicare lo scopo del seguente codice in linguaggio Java evidenziando eventuali errori commessi dal programmatore:

```
class Alfa {
  protected int i, j;

  private void setij(int i, int j)
  {
    i = this.i;
    j = this.j;
  }
}

class Beta extends Alfa {
  public int q;

  void div() {
    q = i*j;
  }
}

class Test {
  public static void main (String args[])
  {
    Beta b = new Beta();
```

```
        b.setij(10, 5);
        b.div();
        System.out.println("Il quoziente è:
                            " + b.q);
    }
}
```

33 Si intende realizzare una gerarchia di classi per rappresentare e gestire un patrimonio immobiliare composto da abitazioni, ville e appartamenti. Le caratteristiche di un'abitazione da impostare in fase di inizializzazione sono il numero di stanze, la superficie, l'indirizzo e la città. Le ville sono caratterizzate inoltre dal numero di piani, dalla superficie del giardino e dal fatto di avere o meno la piscina. Un appartamento è caratterizzato dal piano a cui è situato, dal fatto che sia raggiungibile o meno tramite ascensore e dal numero dei terrazzi. Scrivere il codice Java necessario all'implementazione della gerarchia di classi descritta fornendo per Abitazioni, Ville e Appartamenti i relativi costruttori e i metodi che permettono di visualizzare le caratteristiche citate. Definire mediante un diagramma UML le classi che realizzano la gerarchia descritta valutando l'opportunità di utilizzare una o più classi astratte. Implementare in linguaggio Java le classi progettate specificando costruttori e metodi di accesso agli attributi e sovrascrivendo opportunamente i metodi *toString* ed *equals*. Codificare una classe *Test* il cui metodo *main* istanzi oggetti corrispondenti alle varie tipologie di abitazioni e invochi ciascuno dei metodi definiti almeno una volta.

34 A SCUOLA DI LAVORO Si intende realizzare una gerarchia di classi per rappresentare e gestire i seguenti tipi di oggetti: PC fissi, suddivisi in desktop e server, e PC portatili, suddivisi in notebook e palmari. Le caratteristiche generali di interesse sono: il tipo di processore, la dimensione della memoria RAM, la dimensione della memoria di massa, la marca, il modello e il sistema operativo. I PC fissi sono caratterizzati dal tipo di case (grande, medio, piccolo); per i PC fissi di tipo desktop è necessario registrare il tipo di scheda video e di scheda audio, mentre per i PC fissi di tipo server è necessario sapere il numero dei processori e se hanno o meno dischi di tipo RAID. I PC portatili sono caratterizzati dal peso e dalle dimensioni fisiche (altezza, larghezza e profondità), dalle dimensioni del video, dal fatto di avere o meno l'interfaccia di rete wireless; per i PC portatili di tipo notebook è necessario conoscere se dispongono o meno di webcam e, in caso affermativo, la relativa risoluzione; per i PC portatili di tipo palmare, infine, deve essere memorizzata la presenza o meno dell'interfaccia Bluetooth e il tipo di espansione della memoria di massa (SD, mini-SD, micro-SD, ...). Definire mediante un diagramma UML le classi che realizzano la gerarchia descritta valutando l'opportunità di utilizzare una o più classi astratte. Implementare in linguaggio Java le classi progettate specificando costruttori e metodi di accesso agli attributi e sovrascrivendo opportunamente i metodi *toString* ed *equals*. Codificare una classe *Test* il cui metodo *main* istanzi oggetti corrispondenti alle varie tipologie di PC e invochi ciascuno dei metodi definiti almeno una volta.

35 Si intende realizzare una gerarchia di classi per rappresentare e gestire i dipendenti di una scuola: docenti, impiegati e impiegati che effettuano straordinari. Ogni docente è caratterizzato dal nominativo, dal sesso, dalla data di nascita, dal numero di ore di docenza e dallo stipendio mensile: queste informazioni devono essere impostate in fase di inizializzazione di un oggetto *docente*. Ogni impiegato è caratterizzato dal nominativo, dal sesso, dalla data di nascita, dal livello e dallo stipendio mensile: queste informazioni devono essere impostate in fase di inizializzazione di un oggetto *impiegato*. Un impiegato che effettua straordinari è un impiegato a cui viene attribuito un certo numero di ore di straordinario mensili e una relativa retribuzione oraria da impostare in fase di inizializzazione: la retribuzione oraria è sempre la stessa per tutti gli impiegati che effettuano straordinari e il loro stipendio integra il pagamento delle ore di straordinario effettuate.

Definire mediante un diagramma UML le classi che realizzano la gerarchia descritta valutando l'opportunità di utilizzare una o più classi astratte. Implementare in linguaggio Java le classi progettate specificando costruttori e metodi di accesso agli attributi e sovrascrivendo opportunamente i metodi *toString* ed *equals*. Codificare una classe *Test* il cui metodo *main* istanzi oggetti corrispondenti alle varie tipologie di dipendenti e invochi ciascuno dei metodi definiti almeno una volta.

36 Il Ministero dell'Istruzione deve commissionare un software per il calcolo dei contributi statali dovuti alle scuole. Un professionista viene incaricato di progettare e implementare la gerarchia di classi che rappresenta le scuole. Durante

un'intervista con il direttore generale del Ministero emerge quanto segue:

- le scuole possono essere: elementari, medie o superiori;
- per ogni scuola è necessario memorizzare il codice alfanumerico, la denominazione, l'indirizzo e la città, il numero di studenti, il numero di classi, il numero di sedi aggiuntive e il numero complessivo di laboratori;
- le scuole elementari hanno diritto a un contributo annuale per ogni studente e per ogni sede aggiuntiva: i contributi valgono oggi 125 € per ogni studente e 9000 € per ogni sede aggiuntiva (i valori potrebbero essere modificati in futuro);
- le scuole medie hanno diritto a un contributo annuale per ogni studente, per ogni laboratorio e per ogni sede aggiuntiva: i contributi valgono oggi 150 € per ogni studente, 1100 € per ogni laboratorio e 9000 € per ogni sede aggiuntiva (i valori potrebbero essere modificati in futuro);
- le scuole superiori sono di tre tipi diversi: licei, tecnici e professionali;
- i licei hanno diritto a un contributo annuale uguale a quello delle scuole medie, escluso il contributo per eventuali sedi aggiuntive;
- i tecnici hanno diritto a un contributo annuale per ogni classe e per ogni indirizzo: i contributi valgono oggi 3500 € per ogni classe e 6000 € per ogni laboratorio (i valori potrebbero essere modificati in futuro);
- i professionali – che hanno diritto anche a contributi regionali – hanno diritto a un contributo statale di 2400 € per ogni classe e di 3000 € per ogni laboratorio (i valori potrebbero essere modificati in futuro).

L'incarico del professionista consiste nel progettare mediante un diagramma UML le classi che rappresentano la situazione descritta dal direttore generale e di implementarle in linguaggio Java. Per verificare il lavoro svolto è necessario realizzare una classe *Test* il cui metodo *main* istanzi oggetti corrispondenti alle varie tipologie di scuole calcolando e visualizzando per ciascuna una sintetica descrizione e il contributo da pagare.

37 A SCUOLA DI LAVORO Un'agenzia di pratiche automobilistiche deve commissionare un software per il pagamento delle tasse annuali sui veicoli. Un professionista viene incaricato di progettare e implementare la gerarchia di classi che rappresentano i veicoli. Durante un'intervista al proprietario dell'agenzia emerge quanto segue:

- i veicoli possono essere motoveicoli o autoveicoli;
- per ogni veicolo è necessario memorizzare la targa, la marca, il modello, l'anno di immatricolazione e il numero di passeggeri consentito oltre al conducente;
- i motoveicoli sono sempre alimentati a benzina e sono caratterizzati dalla potenza espressa in HP: la tassa viene calcolata moltiplicando per la potenza un valore che a oggi vale 1,5 €/HP;
- gli autoveicoli tradizionali possono essere alimentati a benzina o a gasolio e sono caratterizzati dalla potenza espressa in HP: la tassa viene calcolata moltiplicando per la potenza un valore che a oggi vale 2,5 €/HP;
- per gli autoveicoli alimentati a gas, oltre alla potenza è necessario memorizzare il tipo di gas (GPL o metano): questi autoveicoli non pagano nessuna tassa per i primi 5 anni dall'immatricolazione, trascorso questo periodo la tassa viene calcolata moltiplicando per la potenza un valore che a oggi vale 0,5 €/HP per il metano e 0,75 €/HP per il GPL;
- gli autoveicoli alimentati a gas idrogeno pagano una tassa che aumenta di 0,1 €/HP per ogni anno di vita del veicolo a partire da una tassa iniziale pari a 0 €/HP il primo anno di immatricolazione;
- come incentivo governativo gli autoveicoli elettrici non pagano alcuna tassa.

L'incarico consiste nel progettare mediante un diagramma UML le classi che rappresentano la situazione descritta dal proprietario dell'agenzia e di implementarle in linguaggio Java. Per verificare il lavoro svolto è necessario realizzare una classe *Test* il cui metodo *main* istanzi oggetti corrispondenti alle varie tipologie di veicoli visualizzando per ciascuno una breve descrizione delle caratteristiche e la tassa da pagare.

38 Si intendono gestire mediante un programma Java i vagoni che compongono un treno. Per ogni vagone si hanno alcuni attributi fondamentali:

- codice;
- peso a vuoto;
- azienda costruttrice;
- anno di costruzione.

Per i vagoni passeggeri si devono inoltre memorizzare

- classe;
- numero di posti disponibili;
- numero di posti occupati;

mentre per i vagoni merci si devono memorizzare:

- volume di carico;
- peso massimo di carico;
- peso effettivo di carico.

Per la composizione di un treno è fondamentale la gestione del peso dei vagoni, che nel caso dei carri merci è di immediata determinazione, mentre per le carrozze passeggeri deve essere stimato considerando un peso medio per occupante di 65 kg (valore che potrebbe essere necessario modificare).
Dopo aver disegnato il diagramma UML delle classi della soluzione proposta e averlo implementato in linguaggio Java, codificare una classe Java *Treno* con uno o più metodi per l'aggiunta di vagoni: la classe dovrà prevedere un metodo che restituisca il peso complessivo del treno esclusa/e la/e motrice/i.

39 A SCUOLA DI LAVORO Di tutti i membri del personale di un'organizzazione di consulenza sono memorizzati nel sistema informatico della stessa i seguenti dati:

- codice;
- cognome;
- nome;
- anno di assunzione o inizio collaborazione.

I membri del personale dell'organizzazione si suddividono in

- dirigenti;
- funzionari;
- tecnici;

di cui solo i tecnici sono specializzati in una specifica area di competenza (informatica-telecomunicazioni o elettronica-automazione) e possono essere sia interni sia esterni (il personale dipendente dell'organizzazione è classificato come «interno», i professionisti collaboratori come «esterni»).
Si intende realizzare un programma per la stima del costo complessivo di partecipazione a un progetto di alcuni membri del personale a partire dal numero di ore di attività previsto per ciascuno di loro sapendo che i costi orari sono valutati come segue:

- tecnico dell'area informatica/telecomunicazioni: 40 €/ora più – ma solo se interno – 1 €/ora per ogni anno trascorso dall'anno di assunzione;
- tecnico dell'area elettronica/automazione: 50 €/ora più – ma solo se interno – 1 €/ora per ogni anno trascorso dall'anno di assunzione;
- funzionario junior (meno di 10 anni di esperienza a partire dall'anno di assunzione o inizio collaborazione): 70 €/ora;
- funzionario senior (più di 10 anni di esperienza a partire dall'anno di assunzione o inizio collaborazione): 80 €/ora;
- dirigente: sempre 100 €/ora.

Ferme restando le regole di calcolo, gli importi orari devono poter essere modificati. Dopo aver disegnato il diagramma UML delle classi della soluzione proposta e averlo implementato in linguaggio Java, codificare una classe Java denominata *Progetto* con uno o più metodi per l'aggiunta di membri del personale al progetto: la classe dovrà prevedere un metodo che restituisca il costo complessivo relativo al personale per l'intero progetto.

40 A SCUOLA DI LAVORO Una grande catena di autonoleggio deve gestire con un sistema informatico i propri veicoli (vetture e furgoni); per ogni veicolo devono essere memorizzate le seguenti informazioni:

- targa;
- numero di matricola;
- marca;
- modello;
- cilindrata;
- anno di acquisto;
- capacità del serbatoio in litri.

Per le vetture è inoltre necessario memorizzare il numero di posti, mentre per i furgoni deve essere memorizzata la capacità di carico.
Progettare mediante un diagramma UML una gerarchia di classi che consenta di rappresentare adeguatamente la situazione descritta; implementare in linguaggio Java la gerarchia di classi progettata. Sapendo inoltre che i veicoli vengono forniti con il pieno di carburante e che il costo del noleggio è così calcolato:

- autovetture: 50 € al giorno, più 1 € ogni 25 km percorsi, più 2 € per ogni litro di carburante che manca dal pieno al momento della restituzione;
- furgoni: 70 € al giorno, più 1 € ogni 30 km percorsi dopo i primi 100 km, più 2 € per ogni

litro di carburante che manca al pieno al momento della restituzione;

implementare una classe Java che consenta la gestione del noleggio dei mezzi e il calcolo dell'importo da far pagare al noleggiatore in funzione dei parametri descritti.

41 Nel videogioco «La battaglia della terra di mezzo» le forze del Bene sono composte dalle razze degli Uomini, degli Elfi, dei Nani e degli Hobbit, mentre le forze del Male dalle razze degli Orchi, degli Urukhai e dei Sudroni.
Nel gioco ogni singolo personaggio appartiene a una delle razze e, di conseguenza, all'uno o all'altro schieramento e ha un attributo esperienza di combattimento codificato mediante un numero intero compreso tra 1 e 10 variabile in base all'esito dei combattimenti a cui partecipa; dispone inoltre di una forza di combattimento calcolabile mediante le seguenti regole:

Razza	Calcolo della forza in base all'esperienza
Uomini	30 + 6 volte l'esperienza
Elfi	Se l'esperienza è inferiore a 5 la forza è 20 + 3 volte l'esperienza, altrimenti è 80 + 2 volte l'esperienza
Nani	20 + 4 volte l'esperienza
Hobbit	10 + 3 volte l'esperienza
Orchi	Se l'esperienza è inferiore a 5 la forza è 30 + 2 volte l'esperienza, altrimenti è 70 + 3 volte l'esperienza
Urukhai	50 + 5 volte l'esperienza
Sudroni	40 + 5 volte l'esperienza

Oltre ai personaggi delle varie razze il gioco prevede gli eroi: ogni eroe appartiene all'uno o all'altro schieramento ma non a una razza specifica ed è identificato da un nome, da un livello di energia vitale che varia (da 1 a 10) in base alle vicissitudini dell'eroe e da una forza di combattimento che è sempre 50 volte l'esperienza di combattimento più 50 volte l'energia vitale.
Il gioco prevede di gestire i singoli personaggi e i singoli eroi che si aggiungono/eliminano nelle varie fasi: quando il gioco ha termine lo schieramento vincente è quello che totalizza la maggiore forza di combattimento. Si richiede:

- un diagramma UML delle classi che modelli lo scenario descritto;
- il codice Java che implementi le classi del diagramma UML progettato;
- il codice di test che inserisca nel gioco un certo numero di personaggi e di eroi con valori di esperienza (ed energia per gli eroi) casuali e che determini lo schieramento vincitore.

Opzionale. Il gioco si svolge su un territorio suddiviso in caselle appartenenti a una griglia righe/colonne: due personaggi (sono esclusi gli eroi che possono intraprendere combattimenti multipli e subire danni che diminuiscono la propria energia vitale) possono combattere tra di loro solo se si trovano in caselle adiacenti; vince chi ha una forza di combattimento maggiore aumentando la propria esperienza di combattimento di un punto se non è già massima, mentre il personaggio sconfitto viene eliminato dal gioco. Integrare le classi già progettate e implementate con gli attributi e i metodi ritenuti necessari per eseguire i combattimenti.

42 È richiesto un programma di allenamento per gli allievi di un corso di scacchi: ogni pezzo ha un tipo specifico (pedone, torre, cavallo, alfiere, regina e re) e appartiene a uno dei due schieramenti (bianco o nero); inoltre occupa una posizione nella scacchiera individuata da un valore numerico compreso tra 1 e 8 inclusi e un carattere alfabetico compreso tra A e H inclusi. Ai fini dell'allenamento i pezzi possono essere posti sulla scacchiera in un modo qualunque ma, ovviamente, due pezzi non possono occupare la stessa casella (*non* è necessaria una matrice per rappresentare la posizione dei pezzi sulla scacchiera) (FIGURA **10**).

FIGURA **10**

Una stima della situazione di gioco può essere effettuata considerando i seguenti punteggi associati ai vari tipi di pezzo (il re deve essere sempre presente, quindi il suo punteggio è inutile a questo scopo):

Pedone	1
Torre	5
Cavallo	3
Alfiere	3
Regina	10

Il programma di allenamento deve consentire di stimare la situazione di gioco fornendo il punteggio per il bianco e per il nero e di aggiornare la posizione dei pezzi (senza controllarne la correttezza, se non limitatamente al fatto che due pezzi non devono occupare la stessa casella della scacchiera), oltre che aggiungere/rimuovere i pezzi dal gioco. Si richiede:

- un diagramma UML delle classi che modelli lo scenario descritto in modo coerente con un design OO;
- il codice Java che implementi le classi del diagramma UML progettato gestendo le opportune eccezioni;
- il codice di test che inserisca nel gioco un certo numero di pezzi e che simuli vari spostamenti ed eliminazioni visualizzando l'elenco finale dei pezzi sulla scacchiera e il punteggio relativo al bianco e al nero.

Opzionale. Integrare la progettazione e il codice in modo che non sia possibile inserire nella scacchiera un numero di pezzi superiore a quello consentito per ciascun tipo e schieramento (8 pedoni, 2 torri, 2 cavalli, 2 alfieri, 1 regina, 1 re). Per una soluzione OO può risultare utile sapere che ogni oggetto Java eredita dalla classe *Object* il metodo *getClass* che restituisce la classe di cui è istanza.

CASI PRATICI

43 A SCUOLA DI LAVORO Realizzare un'applicazione Java per la gestione di un garage pubblico secondo le specifiche illustrate nella bozza di diagramma UML di FIGURA 11 da completare.

Il garage ha al massimo 25 posti, ognuno dei quali è identificato da un numero di posizione: per motivi di capienza possono entrare solo automobili, furgoni e motociclette. Dopo aver progettato e implementato la gerarchia delle classi *Veicolo*, *Furgone*, *Autovettura* e *Motocicletta* implementare, prevedendo le opportune eccezioni, la classe *Garage* i cui metodi devono:

- gestire l'arrivo di un nuovo veicolo nel garage assegnando un posto di cui restituire il numero di posizione e registrando ora e minuti di arrivo;
- gestire l'uscita di un nuovo veicolo dal garage liberando il posto occupato fornito come parametro e restituendo l'importo da pagare calcolato in base alla seguente tabella (ora e minuti di uscita sono forniti come parametro):

Furgone	2 € per ora o frazione
Autovettura	1,5 € per ora o frazione
Motocicletta	1 € per ora o frazione

- esportare ai fini della visualizzazione la situazione corrente dei posti del garage indicando i posti liberi e le informazioni relative ai veicoli per quelli occupati.

La classe *Garage* deve essere dotata di un metodo *main* che – mediante un menu interattivo – deve consentire di simulare tutte le operazioni di gestione del garage sopra descritte.

44 Il proprietario di un pub ha deciso di introdurre un sistema informatizzato per la raccolta e la gestione delle ordinazioni dei clienti dotando ogni cameriere di un palmare collegato senza fili al sistema di raccolta e gestione delle ordinazioni. D'inverno il pub ha 25 tavoli nei locali interni, mentre d'estate può utilizzare la piazza antistante per altri 15 tavoli. Il servizio ai tavoli prevede la raccolta delle ordinazioni che fanno riferimento a bevande alcoliche o analcoliche e snack presenti sul menu

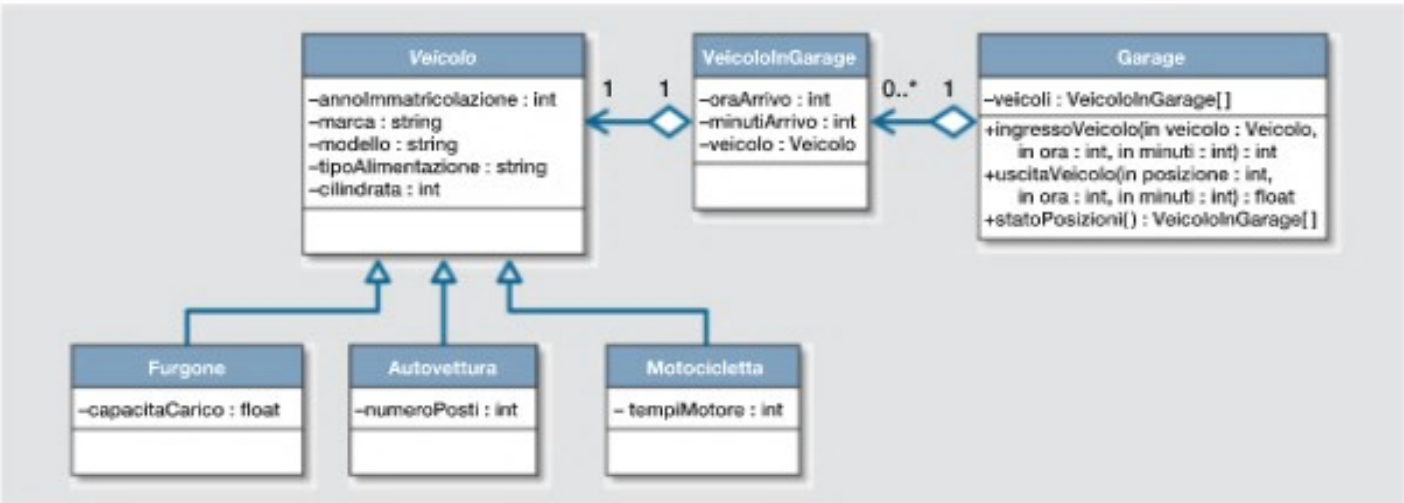


FIGURA 11

da parte dei camerieri e la successiva consegna delle bevande e degli snack ordinati. Raccolta l'ordinazione, il cameriere la consegna al bancone dove uno dei baristi prende un'ordinazione dalla lista di quelle da servire (rispettando l'ordine temporale di consegna) e prepara le bevande e gli snack indicati su un vassoio che appoggia sul bancone a disposizione del cameriere: il cameriere preleva le bevande e gli snack e li consegna ai clienti. Prima di lasciare il pub il cliente passa dalla cassa e comunica al cassiere il numero del suo tavolo, ottenendo il conto che è calcolato utilizzando i prezzi indicati sul menu. Si richiede:

- un diagramma UML delle classi che rappresenti il progetto di una possibile soluzione informatica al problema esposto;
- l'implementazione in linguaggio Java del diagramma UML delle classi;
- una classe di test dotata di un metodo *main* che consenta di effettuare le azioni associate agli attori del diagramma UML dei casi d'uso (FIGURA 12).

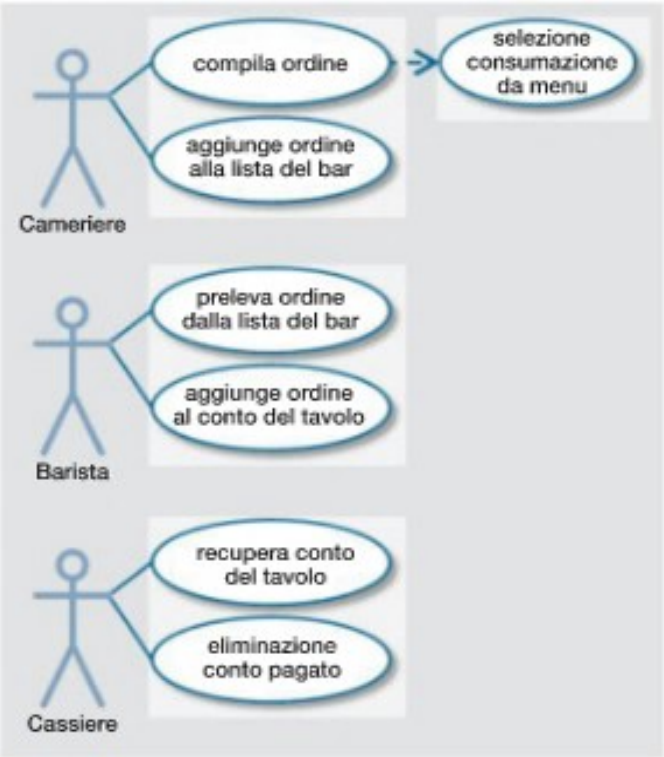


FIGURA 12

45 A SCUOLA DI LAVORO Un'officina meccanica dispone di un magazzino per le parti di ricambio. Ogni parte è catalogata mediante un codice identificativo univoco e una descrizione testuale; una parte può essere complessa (per esempio un motore) o elementare (per esempio una vite): una parte complessa è costituita da altre parti complesse (per esempio il sistema di alimentazione) e/o elementari (per esempio un iniettore). Il catalogo deve essere organizzato a partire dai diversi tipi di auto – definiti da marca, modello e anno di produzione – e deve consentire la ricerca di una specifica parte a partire dal tipo di auto navigando attraverso le sottoparti complesse fino alla parte complessa o elementare desiderata, ma le auto non devono essere considerate parti presenti in magazzino: devono essere infatti inserite nel catalogo esclusivamente come elementi di classificazione. Allo scopo di gestire il magazzino per ogni parte complessa o elementare deve essere memorizzata la quantità presente, che deve essere decrementata quando si prelevano le singole parti per utilizzarle e incrementata quando si acquistano nuove parti dai fornitori; per ogni parte deve anche essere specificata una scorta minima sotto la quale è necessario effettuare un acquisto per reintegrare la scorta. Una parte complessa o elementare può essere prelevata dal magazzino solo se non è sottoparte di una parte complessa (non è per esempio possibile prendere una vite da un motore). Si richiede:

- una rappresentazione mediante un diagramma UML delle classi di una possibile soluzione al problema descritto;
- l'implementazione in linguaggio Java del diagramma UML delle classi progettato prevedendo di realizzare le seguenti funzionalità:
  a) ricerca di una parte complessa o elementare a partire dal codice con indicazione del tipo di auto e delle eventuali parti complesse di cui è sottoparte (per esempio Fiat Cinquecento → motore → sistema di alimentazione → iniettore);
  b) ricerca di una parte complessa o elementare a partire dalla descrizione testuale e dal tipo di auto con indicazione delle eventuali parti complesse di cui è sottoparte;
  c) prelievo di una parte complessa o elementare dal magazzino di cui è noto il codice;
  d) aggiunta al magazzino di nuove parti complesse o elementari di cui è noto il codice;
  e) produzione dell'elenco di parti la cui quantità presente in magazzino è inferiore alla scorta minima prevista;
  f) memorizzazione persistente del catalogo in un file con caricamento e salvataggio dell'intero catalogo.
- la realizzazione di un'interfaccia uomo-macchina a riga di comando per il test di tutte le funzionalità di cui al punto precedente.

built-in
incorporato

drawback
inconveniente

goal
obiettivo

inheritance
ereditarietà

leap-year
anno bisestile

relationship
relazione

trivial
triviale, banale

to underline
sottolineare

Chapter 1 Software Engineering

[...]

1.2 Program Design

[...]

Inheritance

The object-oriented paradigm provides a powerful reuse tool called inheritance, which allows programmers to create a new class that is a specialisation of an existing class. In this case, the new class is called a subclass of the existing class, which in turn is the superclass of the new class.

A subclass "inherits" features from its superclass. It adds new features, as needed, related to its specialisation. It can also redefine inherited features as necessary. Contrary to the intuitive meaning of super and sub, a subclass usually has more variables and methods than its superclass. "Super" and "sub" refer to the relative positions of the classes in a hierarchy. A subclass is below its superclass, and a superclass is above its subclasses.

Suppose we already have a **Date** class [...], and we are creating a new application to manipulate **Date** objects. Suppose also that in the new application we are often required to "increment" a **Date** variable – to change a **Date** variable so that it represents the next day. For example, if the **Date** object represents 7/31/2001, it would represent 8/1/2001 after being incremented. The algorithm for incrementing the date is not trivial, especially when you consider leap-year rules. But in addition to developing the algorithm, we must address another question: where to implement the algorithm.

There are several options:

- Implement the algorithm within the new application. The code would need to obtain the month, day, and year from the **Date** object using the observer methods, calculate the new month, day, and year, instantiate a new **Date** object to hold the updated month, day, and year, and assign it to the same variable. This might appear to be a good approach, since it is the new application that requires the new functionality. However, if future applications also need this functionality, their programmers have to reimplement the solution for themselves. This approach does not support our goal of reusability.
- Add a new method, called **increment**, to the **Date** class. The code would use the incrementing algorithm to update the month, year, and day values of the current object. This approach is better than the previous approach because it allows any future programs that use the **Date** class to use the new functionality.

However, this also means that every application that uses the **Date** class can use this method. In some cases, a programmer may have chosen to use the **Date** class because of its built-in protection against changes to the object variables.

Such objects are said to be immutable. Adding an increment method to the **Date** class undermines this protection, since it allows the variables to be changed.

- Use inheritance. Create a new class, called **IncDate**, that inherits all the features of the current **Date** class, but that also provides the increment method. This approach resolves the drawbacks of the previous two approaches. We now look at how to implement this third approach.

We often call the inheritance relationship an is-a relationship. In this case we would say that an object of the class **IncDate** is also a **Date** object, since it can do anything that a **Date** object can do – and more. This idea can be clarified by remembering that inheritance typically means specialisation. **IncDate** is a special case of **Date**, but not the other way around. To create **IncDate** in Java we would code:

```
public class IncDate extends Date
{
public IncDate(int newMonth, int newDay, int newYear)
// Initialises this IncDate with the parameter values
1.2 Program Design | 19
{
super(newMonth, newDay, newYear);
}
public void increment()
// Increments this IncDate to represent the next day, i.e.,
// this = (day after this)
// For example if this = 6/30/2003 then this becomes 7/1/2003
{
// Increment algorithm goes here
}
}
```

Note that sometimes in code listings we emphasise the sections of code most pertinent to the current discussion by underlining them.

Inheritance is indicated by the keyword **extends**, which shows that **IncDate** inherits from **Date**. It is not possible in Java to inherit constructors, so **IncDate** must supply its own. In this case, the **IncDate** constructor simply takes the month, day, and year parameters and passes them to the constructor of its superclass; it passes them to the **Date** class constructor using the **super** reserved word.

The other part of the **IncDate** class is the new increment method, which is classified as a transformer method, because it changes the internal state of the object. The **increment** method changes the object's day and possibly the month and year values. The **increment** transformer method is invoked through the object that it is to transform.

For example, the statement

```
ourDate.increment();
```

transforms the **ourDate** object.
Note that we have left out the details of the **increment** method since they are not crucial to our current discussion.

A program with access to both of the date classes can now declare and use both **Date** and **IncDate** objects. Consider the following program segment. (Assume output is one of Java's PrintWriter file objects.)

```
Date myDate = new Date(6, 24, 1951);
IncDate aDate = new IncDate(1, 11, 2001);
output.println("mydate day is: " + myDate.dayIs());
output.println("aDate day is: " + aDate.dayIs());
aDate.increment();
output.println("the day after is: " + aDate.dayIs());
```

This program segment instantiates and initialises **myDate** and **aDate**, outputs the values of their days, increments **aDate** and finally outputs the new day value of **aDate**.

[...]

[Nell Dale, Daniel T. Joyce, Chip Weems, *Object-Oriented Data Structures using Java*, Jones & Bartlett Learning, 2002]

QUESTIONS

a **What is inheritance?**
b **How is inheritance implemented in Java?**
c **Which are the advantages of inheritance?**
d **What is the relationship between a superclass and a subclass?**

to behave
comportarsi

habit
abitudine

mammal
mammifero

meal
pasto

primate
primati
(uomini, scimmie)

rodent
roditore

Chapter 2 The Essence of Objects

[...]

Polymorphism

For a given class hierarchy, it is possible for different subclasses to be derived from a common superclass. Each of the subclasses can override and extend the default properties of the superclass differently. Polymorphism is a characteristic of inheritance that ensures that instances of such subclasses behave correctly.

When a subclass overrides a default method, it uses the same name defined in the superclass. If the behaviour of the default method is adequate, a given subclass does not need to override the method, even if other subclasses do. The derived method can implement completely new behaviour, or use the default method while extending it with additional behaviours.

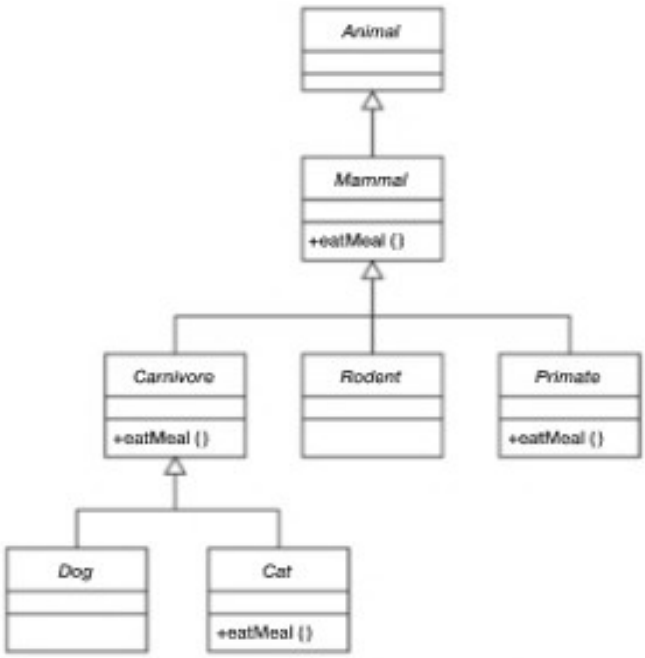


Figure 2.5 – Polymorphism means a Cat uses its own eatMeal, a Dog uses the Carnivore eatMeal, and a Rodent uses the Mammal eatMeal.

The figure shows an **Animal** hierarchy. Since all mammals need to eat, there would likely be a general method defined by the **Mammal** class to handle eating, called **eatMeal**, for example. While all **Mammal** objects might share some eating behaviours that are defined in the general **Mammal eatMeal** method, some would require some specialised eating behaviours that are different than other **Mammal** objects. The eating habits of a **Dog** are different than a **Cat**, which are different than a **Rodent**, which are different than a **Primate**. Thus, each subclass definition can include an **eatMeal** method that implements the specialised eating for that subclass. Not all subclasses need implement a specialised method if the superclass method is satisfactory. In figure 2.5, the **Dog** uses the more general **Carnivore eatMeal**, while the **Cat** has its own specialised **eatMeal**. The **Rodent** uses the general **Mammal eatMeal**. Since all animals may not need an **eatMeal**, there is no **eatMeal** method defined by the **Animal** class.

Note that all these methods have the *same* name, **eatMeal**, even though they implement different behaviours.

If the system were to process a mixed list of different **Mammal** objects, then it would need to use the appropriate **eatMeal** method for each **Mammal**. For example, if the **Mammal** instance were a **Dog**, then the **eatMeal** method defined for the class **Carnivore** would need to be used, and not the **eatMeal** for a **Rodent**.

Polymorphism is what allows the appropriate method for any given object to be used automatically. Polymorphism goes hand in hand with inheritance and classes derived from a common superclass. The mechanism that allows polymorphism to work is **dynamic binding**.

The actual binding of a call to a specific method is delayed until runtime. Only then can the class a particular object instance belongs to be determined, and the correct method from that class then called.

Polymorphism almost seems like magic. It can be difficult to really believe that the proper **eatMeal** will be used for each object.

Fortunately, polymorphism is easier to use than it is to understand completely, and you won't have to think about it explicitly most of the time. Using it comes automatically, and seems a natural part of using objects with inheritance.

dynamic binding Definition: bound at run time.

polymorphism Polymorphism is what allows the appropriate method for any given object to be used automatically.
Polymorphism goes hand in hand with inheritance and classes derived from a common superclass. Polymorphism is supported by dynamic binding of an object to the appropriate method.

[Bruce E. Wampler, Ph.D, *The Essence of Object-Oriented Programming with Java and UML*, Addison-Wesley Professional, 2002]

QUESTIONS

a What is polymorphism?
b What is the relationship between polymorphism and inheritance?
c Why does polymorphism almost seems like magic?
d What is dynamic binding?

A6

Tipi generici e collezioni nel linguaggio Java

La bozza di diagramma UML delle classi di FIGURA 1 rappresenta in forma semplificata l'architettura software di un sito web che commercializza libri, CD audio e film in DVD (la classe *Prodotto* costituisce una loro generalizzazione astratta): il catalogo raccoglie tutti i prodotti disponibili, mentre un carrello comprende i soli prodotti che un cliente intende acquistare.

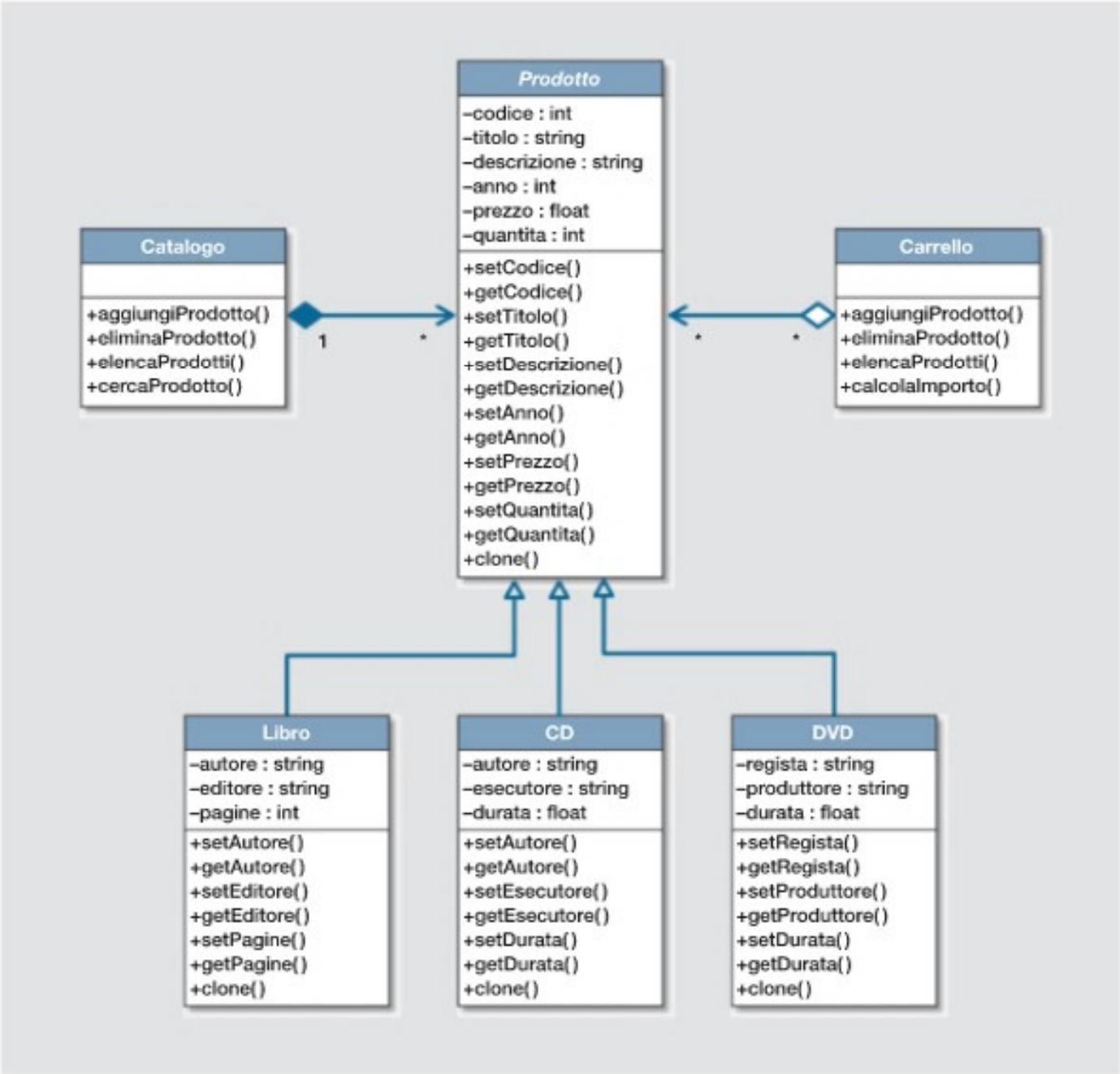


FIGURA 1

Non è difficile implementare in linguaggio Java la gerarchia di classi che derivano dalla classe astratta *Prodotto*, ma quali strutture dati utilizzare per le «collezioni» di prodotti che devono essere gestite dalle classi *Catalogo* e *Carrello*? Potenzialmente il catalogo dovrà contenere decine di migliaia di prodotti, mentre il carrello al massimo alcune decine, ma per entrambe il ricorso a un vettore risulta – data la dinamicità della gestione che prevede eliminazioni oltre che aggiunte – problematico. Sarebbero senz'altro più adatte le strutture dati dinamiche prese in esame nei capitoli precedenti – liste, alberi, tabelle *hash* – ma la loro corretta gestione non è di semplice implementazione.

Il linguaggio di programmazione Java, come tutti i linguaggi di programmazione OO, rende disponibili specifiche classi predefinite per la gestione di contenitori – **collezioni** nella terminologia Java – di oggetti. Le classi delle collezioni Java espongono tutti i metodi necessari per essere utilizzate in modo semplice e sicuro.

ESEMPIO

La classe *Carrello* potrebbe essere implementata utilizzando un oggetto istanza della classe *LinkedList* (una lista concatenata) disponibile nel *package java.util*:

```
import java.util.*;

public class Carrello {
  LinkedList<Prodotto> prodotti; // lista di prodotti

  // costruttore
  public Carrello() {
    prodotti = new LinkedList<Prodotto>();
  }

  // aggiunta di un nuovo prodotto al carrello
  public void aggiungiProdotto(Prodotto prodotto) {
    prodotti.add(prodotto.clone());
  }

  // eliminazione di un prodotto dal carrello
  public void eliminaProdotto(int codice) {
    Iterator<Prodotto> i = prodotti.iterator();
    Prodotto prodotto;

    while (i.hasNext()) {
      prodotto = i.next();
      if (prodotto.getCodice() == codice)
        i.remove();
    }
  }

  // restituzione di un vettore contenente i prodotti del carrello
  public Prodotto[] elencaProdotti() {
    Prodotto[] tmp = (Prodotto[])prodotti.toArray();

    return tmp;
  }
```

```
    // calcolo del costo totale dei prodotti del carrello
    public double calcolaImporto() {
    Iterator<Prodotto> i = prodotti.iterator();
    double totale = 0.0;
    Prodotto prodotto;

    while (i.hasNext()) {
      prodotto = i.next();
      totale += prodotto.getPrezzo()*prodotto.getQuantita();
    }
    return totale;
  }
}
```

La dichiarazione della classe collezione di tipo *LinkedList* prevede la definizione del tipo degli oggetti che costituiscono gli elementi della lista – in questo caso il tipo astratto *Prodotto* – tra i simboli «<» e «>»: le classi collezione predefinite del linguaggio Java prevedono infatti un **parametro di tipo** che deve essere istanziato contestualmente alla dichiarazione. Nell'invocazione del costruttore della classe collezione effettuata per istanziare la collezione stessa può essere omesso il tipo degli elementi giustapponendo i simboli «<» e «>».

Per prendere in esame sequenzialmente tutti gli elementi della lista – nell'esempio precedente ciò è necessario nei metodi *eliminaProdotto* e *calcolaImporto* – è necessario utilizzare un oggetto **iteratore** che espone metodi specifici per effettuare questo tipo di visita: anche la dichiarazione di un iteratore richiede la definizione del tipo degli oggetti elementi della collezione che deve essere visitata. Dato che il tipo può essere inferito dall'oggetto collezione può essere omessa la sua esplicitazione giustapponendo i simboli «<» e «>».

ESEMPIO La classe *Catalogo* richiede ricerche frequenti a partire dal codice in un insieme potenzialmente grande di prodotti; una valida scelta per la sua implementazione potrebbe essere il ricorso a un oggetto istanza della classe *HashMap* (una tabella *hash*) disponibile nel *package java.util*:

```
import java.util.*;

// eccezioni sollevate dai metodi della classe Catalogo
class TroppiProdotti extends Exception {
}
class ProdottoEsistente extends Exception {
}
class ProdottoInesistente extends Exception {
}

public class Catalogo {
  final static int NUMERO_MASSIMO_PRODOTTI = 100000;
```

```
    HashMap<Integer, Prodotto> prodotti; // tabella hash di prodotti

    // costruttore
      public Catalogo() {
        prodotti = new HashMap<Integer, Prodotto>(NUMERO_MASSIMO_PRODOTTI);
    }

    // aggiunta di un nuovo prodotto al catalogo
    public void aggiungiProdotto(Prodotto prodotto)
                throws TroppiProdotti, ProdottoEsistente {
      if (prodotti.size() >= NUMERO_MASSIMO_PRODOTTI)
        throw new TroppiProdotti();
      if (prodotti.containsKey(prodotto.getCodice()))
        throw new ProdottoEsistente();
      prodotti.put(prodotto.getCodice(), prodotto.clone());
    }

    // eliminazione di un prodotto dal catalogo a partire dal codice
    public void eliminaProdotto(int codice) throws ProdottoInesistente {
      if (prodotti.isEmpty() || !prodotti.containsKey(codice))
        throw new ProdottoInesistente();
      prodotti.remove(codice);
    }

    // eliminazione di un prodotto dal catalogo a partire dal titolo
    public void eliminaProdotto(String titolo)throws ProdottoInesistente {
      for (Prodotto prodotto : prodotti.values())
        if (prodotto.getTitolo().equals(titolo))
          eliminaProdotto(prodotto.getCodice());
      throw new ProdottoInesistente();
    }

    // ricerca di un prodotto nel catalogo a partire dal codice
    public Prodotto cercaProdotto(int codice)throws ProdottoInesistente {
      if (prodotti.isEmpty() || !prodotti.containsKey(code))
        throw new ProdottoInesistente();
      return prodotti.get(codice).clone();
    }

    // ricerca di un prodotto nel catalogo a partire dal titolo
    public Prodotto cercaProdotto(String titolo)throws ProdottoInesistente {
      for (Prodotto prodotto : prodotti.values())
        if (prodotto.getTitolo().equals(titolo))
          return prodotto.clone();
      throw new ProdottoInesistente();
    }

    // restituzione di un vettore contenente i prodotti del catalogo
    public Prodotto[] elencaProdotti() {
      Prodotto[] tmp = (Prodotto[])prodotti.values().toArray();

      return tmp;
    }
  }
```

OSSERVAZIONE La dichiarazione della classe collezione di tipo *HashMap* prevede la definizione del tipo degli oggetti collezionati – in questo caso *Prodotto* – e del tipo della chiave a essi collegata, in questo caso un valore intero corrispondente al codice. La tabella *hash* viene organizzata in base alla chiave fornita con ogni prodotto aggiunto al catalogo: le ricerche effettuate in base alla chiave sono immediate ed efficienti[1]; viceversa l'eventuale ricerca in base a un'informazione diversa dalla chiave richiede una scansione completa e sequenziale della collezione che viene svolta ricorrendo a un iteratore. In questo caso l'uso dell'iteratore è implicito nel ciclo utilizzato per prendere in esame i prodotti uno ciascuno:

```
for (Prodotto prodotto : prodotti.values()) {
   ...
   ...
   ...
}
```

Infatti un ciclo *for-each* di questo tipo può essere utilizzato solo su una collezione per la quale sia disponibile un iteratore: l'invocazione del metodo *values* estrae dalla tabella *hash* una collezione di questo tipo.

1. L'efficienza, come abbiamo visto nei capitoli precedenti, dipende dal fattore di carico della tabella che può essere impostato come secondo parametro del costruttore degli oggetti di tipo *HashMap*: il valore di default è 75%.

1 Tipi parametrici e classi generiche in linguaggio Java

Un **tipo parametrico** del linguaggio Java è un identificatore di tipo che nella definizione di una classe – che in questo caso prende il nome di **classe generica** – non viene esplicitato: nella classe generica il tipo parametrico è rappresentato da un identificatore che deve essere istanziato al momento della dichiarazione di un riferimento alla classe.

La definizione di una classe generica elenca i tipi parametrici utilizzati dopo il nome della classe tra i simboli «<» e «>». La dichiarazione di un riferimento avente come tipo una classe generica elenca i tipi effettivamente istanziati dopo il nome della classe tra i simboli «<» e «>»: per istanziare tipi primitivi è necessario impiegare la classe *wrapper* associata. In una classe generica non è possibile creare oggetti o *array* di tipi parametrici.

ESEMPIO La frequente necessità di disporre di coppie di oggetti di tipo eterogeneo può essere facilmente risolta definendo la seguente classe generica con variabili di tipo *T1* e *T2* per i due elementi della coppia:

```
public class Coppia<T1, T2> {
   private T1 primo;
   private T2 secondo;

   public Coppia(T1 primo, T2 secondo) {
      this.primo = primo;
      this.secondo = secondo;
   }
```

Java *generic*

Pur non avendo rappresentato la rivoluzione che è stata l'introduzione dei *template* in C++ (la *Standard Template Library* è la libreria principale del linguaggio), i Java *generic* hanno avuto un impatto importante su molte caratteristiche del linguaggio.
La loro relativamente tarda introduzione ha portato i progettisti di Java a un approccio evolutivo che ne consente un uso parziale tramite i tipi generici *raw*, permettendo di riutilizzare il codice sviluppato nelle versioni precedenti del linguaggio.
Questo paragrafo presenta in modo parziale le sole classi generiche, trascurando il ricorso ai *generic* per scopi diversi come, per esempio, la definizione di metodi generici in classi non generiche, o la definizione di eccezioni generiche.

```
    public T1 getPrimo() {
      return primo;
    }

    public void setPrimo (T1 primo) {
      this.primo = primo;
    }

    public T2 getSecondo() {
      return secondo;
    }

    public void setSecondo(T2 secondo) {
      this.secondo = secondo;
    }

    public String toString() {
      return "("+primo.toString()+";"+secondo.toString()+")";
    }
  }
```

Volendo, per esempio, utilizzare valori numerici con associata la propria unità di misura è possibile dichiarare oggetti utilizzando la seguente sintassi:

```
Coppia<Double, String> c1 = new Coppia<Double,
                                       String>(1.1, "Kg");
```

Se invece la coppia serve per associare la frequenza di una misurazione al valore della misura risulta utile la seguente dichiarazione:

```
Coppia<Double, Integer> c2 = new Coppia<Double,
                                        Integer>(0.9E-3, 12);
```

Gli oggetti così istanziati possono essere utilizzati come un qualsiasi oggetto del linguaggio Java:

```
...
c2.setPrimo(99.99); // accetta come parametro un Double
c2.setSecondo(101); // accetta come parametro un Integer
double valore = c2.getPrimo(); // restituisce un valore Double
int frequenza = c2.getSecondo(); // restituisce un valore Integer
c1.setPrimo(valore*frequenza); // accetta come parametro un Double
c1.setSecondo("m"); // accetta come parametro un oggetto String
...
```

Naturalmente il controllo di tipo effettuato dal compilatore è specializzato in base alla dichiarazione dei singoli oggetti ed è quindi differenziato per *c1* e *c2*.

Nelle dichiarazioni di oggetti di classe *Coppia* nell'esempio precedente i tipi di *c1* e *c2* sono diversi: il primo è *Coppia<Double, String>*, mentre il secondo è *Coppia<Double, Integer>*.
Entrambi gli oggetti sono però istanza della stessa classe *Coppia*.

L'utilità delle classi generiche è evidente nel caso in cui si devono progettare classi aventi la funzione di contenitori di oggetti. La classe *Coda* introdotta in un capitolo precedente risultava utilizzabile esclusivamente per gli oggetti istanza di una specifica classe, o delle sue derivate; la seguente classe generica *Coda<T>* consente di dichiarare contenitori di un qualsiasi tipo *T*:

```java
// classe parametrica Nodo:
// l'attributo "info" ha tipo generico T
class Nodo<T> {
  private T info;
  private Nodo<T> link;

  public Nodo(T info) {
    this.info = info;
    link = null;
  }

  public void setInfo(T info) {
    this.info = info;
  }
  public T getInfo() {
    return info;
  }

  protected void setLink(Nodo<T> link) {
    this.link = link;
  }
  public Nodo<T> getLink() {
    return link;
  }
}

// classe parametrica Coda: gli attributi "head" e "tail"
// hanno come tipo generico la classe parametrica Nodo (Nodo<T>)
public class Coda<T> {
  private Nodo<T> head;
  private Nodo<T> tail;

  public Coda() {
    head = null;
    tail = null;
  }

  private Nodo<T> creaNodo(T info, Nodo<T> link) {
    Nodo<T> p = new Nodo<>(info);
    p.setLink(link);
    return p;
  }

  // inserimento di un elemento di tipo parametrico
  // nella coda
  public void enqueue(T info) {
    Nodo<T> p = creaNodo(info, null);
    if (head == null) {
      tail = p;
      head = tail;
    }
    else {
      tail.setLink(p);
      tail = p;
```

```
    }
  }

  // estrazione di un elemento di tipo parametrico
  // dalla coda
  public T dequeue() {
    Nodo<T> p;
    if (head == null)
      return null;
    p = head;
    head = head.getLink();
    if (head == null)
      tail = null;
    return p.getInfo();
  }
}
```

Nella classe *Coda<T>* i riferimenti locali, i parametri e i risultati dei metodi che hanno come tipo un nodo della lista che implementa la coda sono definiti come riferimenti di tipo *Nodo<T>*, cioè della classe generica *Nodo* specializzata per il tipo parametrico *T* che viene specificato al momento della dichiarazione di un riferimento a un oggetto di tipo *Coda*.

ESEMPIO Avendo in precedenza definito una classe *Persona*, è possibile creare una coda di oggetti di tipo *Persona* utilizzando la classe generica *Coda<T>*:

```
...
Persona p1 = new Persona("Bianchi Giovanni", 'M');
Persona p2 = new Persona("Rossi Marta", 'F');
Coda<Persona> coda = new Coda<>();
coda.enqueue(p1);
coda.enqueue(p2);
...
Persona p = coda.dequeue();
...
```

Ma la stessa classe generica *Coda<T>* può essere utilizzata per gestire una coda di oggetti di classe *Punto* se questa classe è stata definita:

```
...
Punto p1 = new Punto(0., 1.);
Punto p2 = new Punto(1., 0.);
Coda<Punto> coda = new Coda<>();
coda.enqueue(p1);
coda.enqueue(p2);
...
Punto p = coda.dequeue();
...
```

OSSERVAZIONE Nelle versioni del linguaggio meno recenti Java non disponeva dei tipi generici: per definire classi contenitore di tipo non vincolato si era soliti utilizzare attributi e parametri di tipo *Object* che garantiscono la possibilità di riferire un qualsiasi oggetto Java. Una possibile implementazione di una coda che adotta questa soluzione è la seguente:

```
// classe Nodo: l'attributo "info" ha tipo Object
class Nodo {
   private Object info;
   private Nodo link;

   public Nodo(Object info) {
      this.info = info;
      link = null;
   }

   public void setInfo(Object info) {
      this.info = info;
   }
   public Object getInfo() {
      return info;
   }

   protected void setLink(Nodo link) {
      this.link = link;
   }
   public Nodo getLink() {
      return link;
   }
}

// classe Coda
public class Coda {
   private Nodo head;
   private Nodo tail;

   public Coda() {
      head = null;
      tail = null;
   }

   private Nodo creaNodo(Object info, Nodo link) {
      Nodo p = new Nodo(info);
      p.setLink(link);
      return p;
   }

   // inserimento di un elemento di tipo indefinito
   // nella coda
   public void enqueue(Object info) {
      Nodo p = creaNodo(info, null);
      if (head == null) {
         tail = p;
         head = tail;
      }
      else {
         tail.setLink(p);
         tail = p
      }
```

```
    // estrazione di un elemento di tipo indefinito
    // dalla coda
    public Object dequeue() {
      Nodo p;
      if (head == null)
        return null;
      p = head;
      head = head.getLink();
      if (head == null)
        tail = null;
      return p.getInfo();
    }
}
```

La classe presenta la stessa flessibilità della classe parametrica:

```
...
Persona p1 = new Persona("Bianchi Giovanni", 'M');
Persona p2 = new Persona("Rossi Marta", 'F');
Coda coda = new Coda();
coda.enqueue(p1);
coda.enqueue(p2);
...
Persona p = (Persona)coda.dequeue();
...
Punto p1 = new Punto(0., 1.);
Punto p2 = new Punto(1., 0.);
Coda coda = new Coda();
coda.enqueue(p1);
coda.enqueue(p2);
...
Punto p = (Punto)coda.dequeue();
...
```

Ricorrendo a questa soluzione è necessario effettuare un *cast* esplicito dell'oggetto che viene estratto dalla coda: quest'operazione è soggetta, nel caso che il programmatore non sia rigoroso nel controllo del tipo degli oggetti inseriti nella coda, a errori catastrofici.
Inoltre, in questo caso le regole del linguaggio Java consentono di inserire nella coda oggetti eterogenei di un tipo qualsiasi, cosa che normalmente non è quella voluta dal programmatore che spesso utilizza una collezione come contenitore per oggetti istanze delle classi di un'unica gerarchia di ereditarietà.

Un tipo ***raw*** è una classe generica utilizzata senza istanziare il tipo parametrico: in questo caso il compilatore non effettua controlli relativi al tipo parametrico che risulta in pratica assimilabile al tipo *Object*.

OSSERVAZIONE I tipi *raw* sono stati introdotti al solo scopo di facilitare l'integrazione di codice Java scritto prima della versione 5 del linguaggio, che ha introdotto le classi generiche, e non dovrebbero essere

In questo caso sono legali unicamente dichiarazioni che istanziano tipi parametrici che derivano dalla classe astratta *Number*:

```
CoppiaNumerica<Byte, Float> c1;
CoppiaNumerica<Double, Integer> c2;
```

Questa classe generica potrebbe essere definita anche per derivazione dalla classe generica *Coppia<T1, T2>* senza dover ridefinire attributi e metodi:

```
public class CoppiaNumerica<T1 extends Number, T2 extends Number> extends Coppia<T1, T2> {
  public CoppiaNumerica(T1 primo, T2 secondo) {
    super(primo, secondo);
  }
}
```

Com'è illustrato nell'esempio precedente, è possibile derivare classi generiche a partire da altre classi generiche e non, anche se le regole da osservare non sempre sono evidenti[2]. Un caso particolare consiste nel derivare una classe non generica istanziando il tipo parametrico (o i tipi parametrici) di una classe generica:

```
public class CoppiaNumericaIntera extends
            Coppia<Integer, Integer> {
  public CoppiaNumerica(Integer primo, Integer secondo) {
    super(primo, secondo);
  }
}
```

2. La comprensione di tutte le clausole limitative che regolano il comportamento delle classi generiche richiede una comprensione del sistema di tipizzazione del linguaggio Java che va al di là dello scopo della presente trattazione.

Non è complesso a questo punto definire un iteratore generico standard per la classe generica *Coda<T>*:

```
public class Iteratore<T> implements Iterator<T> {
  Nodo<T> nodo;

  public Iteratore(Nodo<T> nodo) {
    this.nodo = nodo;
  }

  public boolean hasNext() {
    if (nodo == null)
      return false;
    else
      return true;
  }

  public T next() throws NoSuchElementException {
    if (nodo == null)
      throw new NoSuchElementException();
    T info = nodo.getInfo();
    nodo = nodo.getLink();
    return info;
  }

  public void remove() throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
  }
}
```

L'iteratore viene restituito da uno specifico metodo generico della classe *Coda<T>*:

```
public Iteratore<T> getIteratore() {
  Iteratore<T> i = new Iteratore<T>(head);
  return i;
}
```

ESEMPIO L'iteratore della classe generica *Coda<T>* può essere utilizzato per accedere ai singoli elementi della coda nel seguente modo:

```
...
Coda<Persona> c = new Coda<Persona>();
...
Iteratore<Persona> i = c.getIteratore();
Persona p;

while (i.hasNext()) {
  Persona p = i.next();
  System.out.println(p.toString());
}
...
```

Dovendo definire un metodo statico per calcolare la somma degli elementi di una coda generica fornita come parametro, è necessario garantire che il tipo parametrico su cui la coda è istanziata sia di tipo numerico; è legale scrivere un metodo come il seguente

```
public static double somma(Coda<Number> coda) {
  double s = 0.0;
  Iteratore<Number> i = coda.getIteratore();

  while (i.hasNext()) {
    s = s + i.next().doubleValue();
  }

  return s;
}
```

ma il seguente codice produce errori di tipo in fase di compilazione

```
...
Coda<Integer> c = new Coda<Integer>();
double s;
int i1 = 1;
int i2 = 2;
c.enqueue(i1);
c.enqueue(i2);
s = Coda.somma(c); // errore rilevato in fase di compilazione
...
```

perché la classe *wrapper Integer* del tipo primitivo `int` deriva dalla classe astratta *Number*, ma il tipo generico *Coda<Integer>*, così come il tipo generico *Coda<Double>*, non deriva dal tipo generico *Coda<Number>*, per cui il compilatore non applica il principio di sostituzione tipico del polimorfismo.

Per gestire questo tipo di situazione sono stati introdotti nel linguaggio Java i tipi parametrici **jolly** (*wildcard*):

wildcard	Tipo corrispondente
`<?>`	Tipo senza limiti.
`<? extends S>`	Tipo con limite superiore: deve essere sottotipo di *S*.
`<? super S>`	Tipo con limite inferiore: deve essere super-tipo di *S*.

Le regole del linguaggio Java stabiliscono che il tipo *Coda<Integer>*, così come *Coda<Double>*, sia un sottotipo di *Coda<? extends Number>*, per cui la forma corretta di definizione del metodo somma è la seguente:

```
public static double somma(Coda<? extends Number> coda) {
  double s = 0.0;
  Iteratore<? extends Number> i = coda.getIteratore();

  while (i.hasNext()) {
    s = s + i.next().doubleValue();
  }

  return s;
}
```

OSSERVAZIONE Il tipo *Coda<? extends Number>* identifica una coda di elementi omogenei istanze di una specifica sottoclasse derivata dalla classe astratta *Number*, mentre il tipo *Coda<Number>* identifica una coda di elementi **non** omogenei potenzialmente istanze di varie sottoclassi derivate dalla classe astratta *Number*.

ESEMPIO

Con la forma corretta del metodo *somma* il seguente codice viene compilato ed eseguito senza errori o eccezioni

```
...
Coda<Double> c = new Coda<Double>();
double s;
double d1 = 1.1;
double d2 = 2.2;
c.enqueue(d1);
c.enqueue(d2);
s = Coda.somma(c);
...
```

ma il codice che segue genera un errore in fase di compilazione, questa volta atteso in quanto segnala una reale incompatibilità di tipo:

```
...
Coda<Punto> c = new Coda<Punto>();
double s;
Punto p1 = new Punto(1,1);
Punto p2 = new Punto(2,2);
c.enqueue(p1);
c.enqueue(p2);
s = Coda.somma(c); // errore rilevato in fase di compilazione
...
```

Il tipo jolly con limite superiore (*<? extends S>*) è di solito impiegato per estrarre/copiare (*get*) i valori **da** un oggetto istanza di una classe generica.

ESEMPIO

Il seguente metodo *copyFrom* della classe generica *Coda<T>* copia gli elementi di una coda fornita come parametro che necessariamente deve avere elementi che sono un sottotipo degli elementi della coda stessa:

```
public void copyFrom(Coda<? extends T> coda) {
  Iteratore<? extends T> i = coda.getIteratore();

  while (i.hasNext()) {
    this.enqueue(i.next());
  }
}
```

Questo frammento di codice esemplifica l'uso del metodo *copyFrom*:

```
...
Coda<Number> cn = new Coda<Number>();
Coda<Integer> ci = new Coda<Integer>();
Coda<Double> cd = new Coda<Double>();
ci.enqueue(-1);
ci.enqueue(99);
cd.enqueue(3.1416);
cd.enqueue(1.4142);
cn.copyFrom(ci);
cn.copyFrom(cd);
double s = Coda.somma(cn);
...
```

Il tipo jolly con limite inferiore (*<? super S>*) è invece generalmente impiegato per inserire/copiare (*set*) i valori **in** un oggetto istanza di una classe generica.

ESEMPIO

Il seguente metodo *copyTo* della classe generica *Coda<T>* copia gli elementi in una coda fornita come parametro che necessariamente deve avere elementi che sono un super-tipo degli elementi della coda stessa:

```
public void copyTo(Coda<? super T> coda) {
  Iteratore<T> i = this.getIteratore();
  while (i.hasNext()) {
    coda.enqueue(i.next());
  }
}
```

Questo frammento di codice esemplifica l'uso del metodo *copyTo*:

```
...
Coda<Number> cn = new Coda<Number>();
Coda<Integer> ci = new Coda<Integer>();
Coda<Double> cd = new Coda<Double>();
ci.enqueue(-1);
ci.enqueue(99);
cd.enqueue(3.1416);
cd.enqueue(1.4142);
ci.copyTo(cn);
cd.copyTo(cn);
double s = Coda.somma(cn);
...
```

2 I contenitori del linguaggio Java: le «collezioni»

Il *Collection framework* del linguaggio Java consiste in un insieme di interfacce e di classi definite nel *package java.util* aventi lo scopo di implementare contenitori di oggetti (liste, code, alberi, tabelle, ...) in modo efficace ed efficiente per il programmatore.

Le interfacce fondamentali del *framework* definiscono in modo astratto i tipi di contenitori resi disponibili (TABELLA 1).

TABELLA 1

Interfaccia	Implementazioni fondamentali (classi)	Ordinamento	Duplicazione elementi	Caratteristiche della collezione
List	*ArrayList* *LinkedList*	SÌ	SÌ	Accesso posizionale tramite indice.
Queue	*PriorityQueue*	SÌ	SÌ	Politica FIFO.
Set	*HashSet* *TreeSet*	Dipendente dall'implementazione.	NO	Elementi univoci.
Map	*HashMap* *TreeMap*	Dipendente dall'implementazione.	NO (chiavi)	Uso di chiavi di accesso univoche.

Il diagramma UML di FIGURA 2 schematizza le relazioni di derivazione delle principali interfacce del *Collection framework*; si noti che sono tutte generiche rispetto al tipo parametrico *T* dell'elemento collezionato (le interfacce che definiscono associazioni con una chiave di accesso sono generiche anche rispetto al tipo parametrico *K* della chiave).

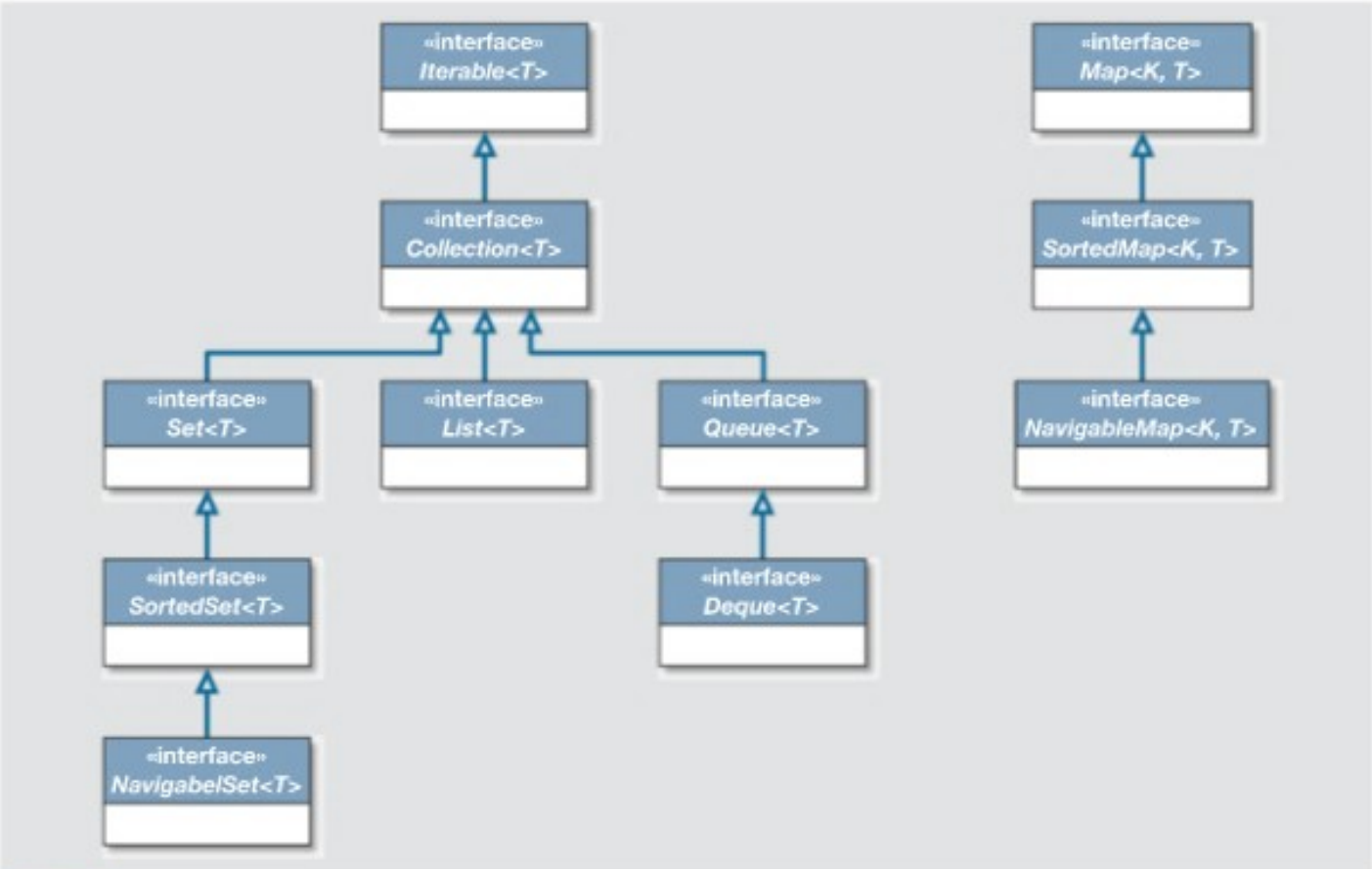


FIGURA 2

Tutte le interfacce che derivano dall'interfaccia *Collection<T>* ereditano l'interfaccia *Iterable<T>*, che richiede l'implementazione di un metodo *iterator* che restituisce un iteratore per gli elementi della collezione. Com'è stato esemplificato nel paragrafo introduttivo, per le istanze delle classi che implementano l'interfaccia *Iterable* è possibile impiegare il ciclo *for-each*, che utilizza l'iteratore in modo implicito per scorrere gli elementi contenuti nella classe. Le classi che implementano l'interfaccia *Map<K, T>* non implementano l'interfaccia *Iterable*, ma dispongono del metodo *values*, che estrae dal contenitore una collezione dotata di iteratore.

ESEMPIO

Una *polyline* è una sequenza ordinata di punti nel piano cartesiano; volendo utilizzare un contenitore di tipo *ArrayList* per la memorizzazione dei punti, il metodo *passaPer* della seguente classe *Polyline* verifica se il punto fornito come argomento appartiene o meno alla sequenza:

```
public class Polyline {
  private ArrayList<Punto> sequenza;

  public Polyline() {
    sequenza = new ArrayList<>();
  }
  ...
  ...
  ...
  public double passaPer(Punto p) {
    for (Punto q : sequenza) // ciclo for-each
      if (q.equals(p))
        return true;
    return false;
  }
  ...
  ...
  ...
}
```

Il metodo *passaPer* poteva essere implementato in modo equivalente utilizzando esplicitamente un oggetto iteratore:

```
public double passaPer(Punto p) {
  Iterator<Punto> i;

  for (i = sequenza.iterator(); i.hasNext();) {
    Punto q = i.next();
    if (q.equals(p))
      return true;
  return false;
}
```

Il secondo diagramma UML schematizza le relazioni di derivazione delle principali classi del *Collection framework*; anche in questo caso sono tutte generiche rispetto al tipo parametrico *T* dell'elemento collezionato e le classi che definiscono associazioni con una chiave di accesso sono generiche anche rispetto al tipo parametrico *K* della chiave (FIGURA 3).

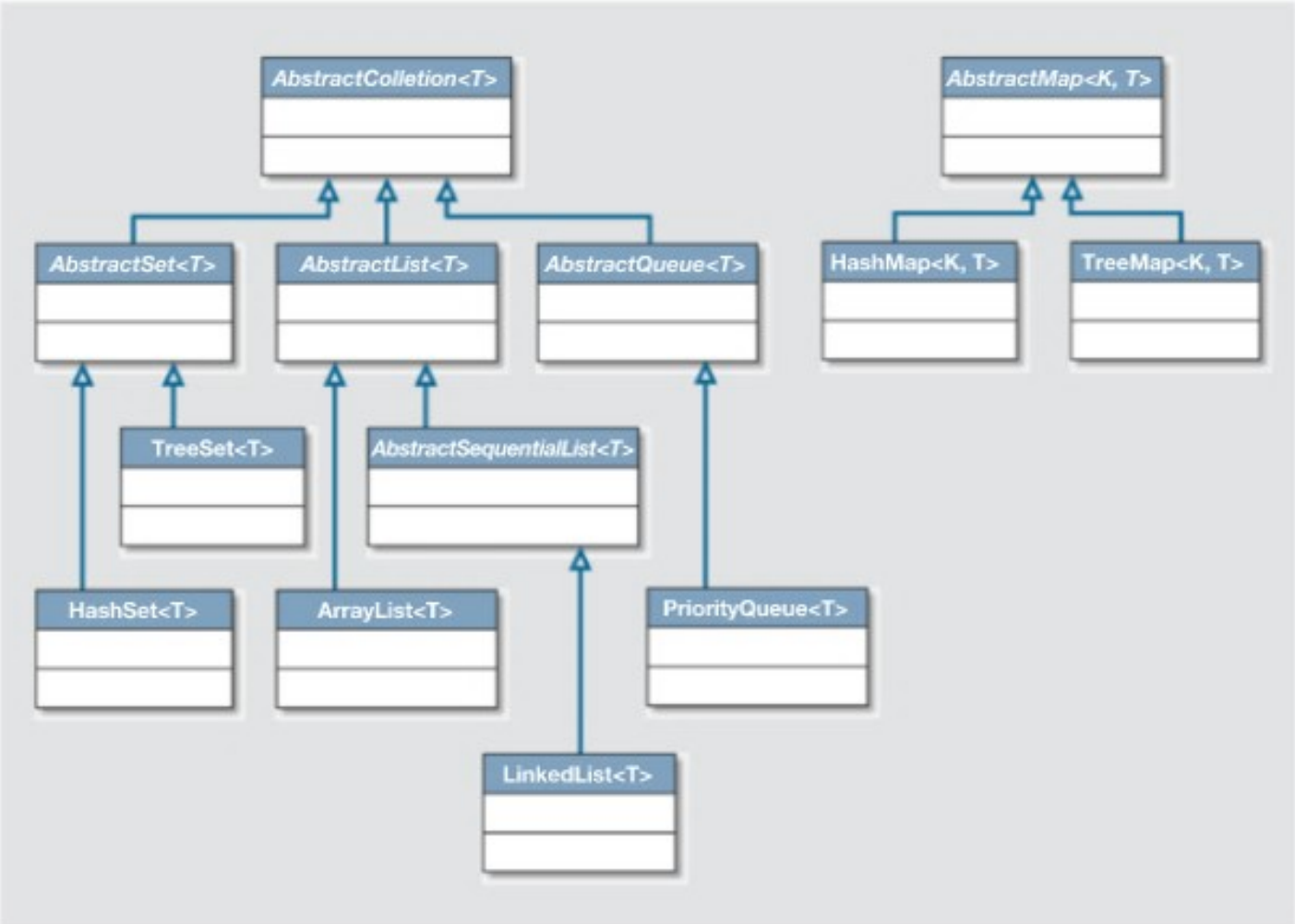


FIGURA 3

Le classi generiche concrete presenti nel diagramma UML di FIGURA 3 (*HashSet<T>*, *TreeSet<T>*, *ArrayList<T>*, *LinkedList<T>*, *PriorityQueue<T>*, *HashMap<K, T>*, *TreeMap<K, T>*, ...) sono quelle effettivamente utilizzate per la realizzazione di contenitori da parte dei programmatori.

I nomi delle classi evidenziano la loro implementazione interna:

- *TreeSet<T>* e *TreeMap<K, T>* sono alberi bilanciati;
- *HashSet<T>* e *HashMap<K, T>* sono tabelle *hash*;
- *ArrayList<T>* è un vettore dinamico;
- *LinkedList<T>* è una lista concatenata.

Come vedremo ogni specifica implementazione ha i suoi punti di forza e di debolezza, in particolare in relazione all'efficienza delle operazioni che si possono effettuare.

Molte classi generiche utilizzabili per realizzare un contenitore impongono agli elementi di tipo parametrico *T* e/o alle chiavi di tipo parametrico *K* un ordinamento. A questo scopo, nel caso di classi definite dal programmatore, è buona norma che esse implementino l'interfaccia *Comparable<T>* definita nel *package java.lang* che richiede l'implementazione del seguente metodo

```
int compareTo(T object);
```

che deve restituire un valore intero negativo, nullo o positivo a seconda che l'oggetto su cui viene invocato sia minore, uguale o maggiore dell'oggetto fornito come parametro.

ESEMPIO

La classe Java *Punto* che segue implementa l'interfaccia *Comparable<T>*; il metodo *compareTo* confronta la distanza dall'origine del punto su cui viene invocato con la distanza dall'origine del punto fornito come parametro:

```
public class Punto implements Comparable<Punto> {
  private double x;
  private double y;

  public Punto(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public Punto(Punto p) {
    this.x = p.getX();
    this.y = p.getY();
  }

  public double getX() {
    return x;
  }
  public void setX(double x) {
    this.x = x;
  }

  public double getY() {
    return y;
  }
  public void setY(double y) {
    this.y = y;
  }

  public String toString() {
    return "(" + this.getX() + ";" + this.getY() + ")";
  }

  public double distanza(Punto p) {
    return Math.sqrt(Math.pow((p.getX() - this.getX()), 2) +
           Math.pow((p.getY() - this.getY()), 2));
  }

  public int compareTo(Punto p) {
    Punto o = new Punto(0.0, 0.0); // origine
    if (this.distanza(o) < p.distanza(o))
      return - 1;
    if (this.distanza(o) > p.distanza(o))
      return 1;
    return 0;
  }
}
```

2.1 Le classi contenitore che implementano le interfacce *Collection<T>* e *List<T>*

L'interfaccia *Collection<T>* definisce i metodi, riportati nella TABELLA 2, che sono comuni a tutte le classi che la implementano (non sono stati riportati i metodi non specifici dell'interfaccia).

TABELLA 2

Firma del metodo	Descrizione della funzionalità
`boolean add(T e);`	Aggiunta di un elemento *e* alla collezione.
`boolean addAll(Collection<? extends T> c);`	Aggiunta di tutti gli elementi della collezione *c* alla collezione.
`void clear();`	Eliminazione di tutti gli elementi della collezione.
`boolean contains(Object e);`	Verifica della presenza di un elemento *e* nella collezione.
`boolean containsAll(Collection<?> c);`	Verifica della presenza di tutti gli elementi di una collezione *c* nella collezione.
`boolean isEmpty();`	Verifica se la collezione è vuota.
`Iterator<T> iterator();`	Restituisce un iteratore per gli elementi della collezione.
`boolean remove(Object e);`	Elimina dalla collezione l'elemento *e*.
`boolean removeAll(Collection<?> c);`	Elimina dalla collezione tutti gli elementi contenuti nella collezione *c*.
`boolean retainAll(Collection<?> c);`	Mantiene nella collezione solo gli elementi presenti anche nella collezione *c*.
`int size();`	Restituisce il numero di elementi presenti nella collezione.
`Object[] toArray();`	Esporta gli elementi della collezione in un *array* di elementi di tipo *Object*.
`<T> T[] toArray(T a[]);`	Esporta gli elementi della collezione in un *array* di elementi di tipo *T*.

OSSERVAZIONE Le modalità di uso – e in alcuni casi di «non uso» – dei parametri di tipo nella definizione dell'interfaccia può sorprendere: i progettisti del linguaggio hanno effettuato le scelte più sicure compatibilmente con le complesse direttive che regolano l'uso dei tipi generici in Java e talvolta questo ha comportato il ricorso al tipo *Object*, o al jolly di tipo non limitato *<?>*. In particolare il metodo *toArray* ha due versioni in *overloading*: la prima non è sicura dal punto di vista del tipo (restituisce un vettore di oggetti di tipo *Object* su cui sarà necessario effettuare un *cast*), la seconda lo è, ma richiede di fornire l'*array* stesso (che sarà eventualmente ridimensionato) come parametro[3].

3. La necessità di fornire l'*array* risultato come parametro è dettata da problematiche di «reificazione» dei tipi generici che esula dallo scopo della presente trattazione.

Prendiamo in esame, tra tutte le classi che implementano l'interfaccia *Collection<T>*, quelle che implementano anche l'interfaccia *List<T>*, che aggiunge la definizione dei metodi riportati nella TABELLA 3.

TABELLA 3

Firma del metodo	Descrizione della funzionalità
`boolean add(int i, T e);`	Aggiunta di un elemento *e* alla lista in posizione *i*.
`boolean addAll(int i, Collection<? extends T> c);`	Aggiunta di tutti gli elementi della collezione *c* alla lista a partire dalla posizione *i*.
`T get(int i);`	Restituisce l'elemento in posizione *i* della lista.
`int indexOf(Object e);`	Restituisce la posizione in cui si trova la prima occorrenza dell'elemento *e* nella lista.
`int lastIndexOf(Object e);`	Restituisce la posizione in cui si trova l'ultima occorrenza dell'elemento *e* nella lista.
`ListIterator<T> listIterator();`	Restituisce un iteratore di lista per gli elementi della lista.
`ListIterator<T> listIterator(int i);`	Restituisce un iteratore di lista per gli elementi della lista a partire dalla posizione *i*.
`boolean remove(int i);`	Elimina dalla lista l'elemento in posizione *i*.
`T set(int i, T e);`	Sostituisce nella lista l'elemento nella posizione *i* con l'elemento *e*.
`List<T> subList(int i, int j);`	Restituisce una lista costituita dagli elementi della lista compresi tra le posizioni *i* e *j*.

L'interfaccia *List<T>* rispetto all'interfaccia *Collection<T>* aggiunge la possibilità di gestire gli elementi in base all'indice di posizione. In particolare i metodi *get* e *set* consentono di utilizzare una lista come un *array*. Inoltre il metodo *listIterator* restituisce un oggetto di classe *ListIterator<T>* che aggiunge a *Iterator<T>*, da cui deriva, la possibilità di scorrere la lista in modo bidirezionale e di aggiungere elementi alla lista.

Le classi concrete generiche principali che implementano le interfacce *Collection<T>* e *List<T>* sono *ArrayList<T>* e *LinkedList<T>*: la prima – in analogia a un *array* – prevede una collezione di un numero massimo di elementi che può essere impostato come parametro del costruttore e successivamente modificato invocando il metodo *ensureCapacity*; la seconda è – essendo una vera e propria lista concatenata – indefinitamente estensibile.

Anche se le classi generiche *ArrayList<T>* e *LinkedList<T>* hanno numerosi metodi specifici, esse implementano le stesse interfacce ed espongono di conseguenza un insieme di metodi comuni a entrambe. La scelta del tipo di contenitore corretto dipende quindi fondamentalmente dall'uso che il programmatore intende farne e dalle prestazioni richieste. La TABELLA 4 permette di effettuare la scelta (*N* è il numero di elementi presenti nella collezione)[4].

Array e *ArrayList*

La flessibilità di uso dei contenitori istanza della classe generica *ArrayList<T>* è tale che molti programmatori Java preferiscono utilizzare un contenitore di questo tipo anziché un *array* vero e proprio.
In particolare, la caratteristica di essere dinamicamente ridimensionabili in fase di esecuzione del codice rende particolarmente attraente il ricorso a contenitori di tipo *ArrayList<T>* al posto degli *array*: uno degli svantaggi consiste nell'impossiblità di avere contenitori generici di tipi primitivi, ma la presenza delle classi *wrapper* non rende particolarmente grave questo problema.

4. M. Naftalin e P. Wadler, *Java Generics and Collections*, O'Reilly, 2006.

TABELLA 4

Classe	add	contains	next (Iterator <T>)	remove	get
ArrayList<T>	Tempo costante.	Tempo proporzionale a *N*.	Tempo costante.	Tempo proporzionale a *N*.	Tempo costante.
LinkedList<T>	Tempo costante.	Tempo proporzionale a *N*.	Tempo costante.	Tempo costante.	Tempo proporzionale a *N*.

Se l'uso del contenitore prevede prevalentemente accessi in base alla posizione e un limitato numero di rimozioni, allora la scelta corretta è *ArrayList<T>*. Viceversa, se si hanno frequenti rimozioni e un accesso prevalentemente sequenziale, allora è preferibile *LinkedList<T>* che, oltre ai metodi definiti dall'interfaccia *List<T>*, implementa i tradizionali metodi di accesso a una lista concatenata per l'inserimento/eliminazione di elementi in testa e in coda.
In entrambi i casi le differenze in termini di prestazioni sono evidenti solo per un numero *N* di elementi elevato.

ESEMPIO

Con riferimento alla gerarchia di classi di FIGURA 4, definita nei capitoli precedenti, si intende progettare una classe Java che consenta la gestione dei dipendenti di una scuola permettendo le seguenti operazioni di base:

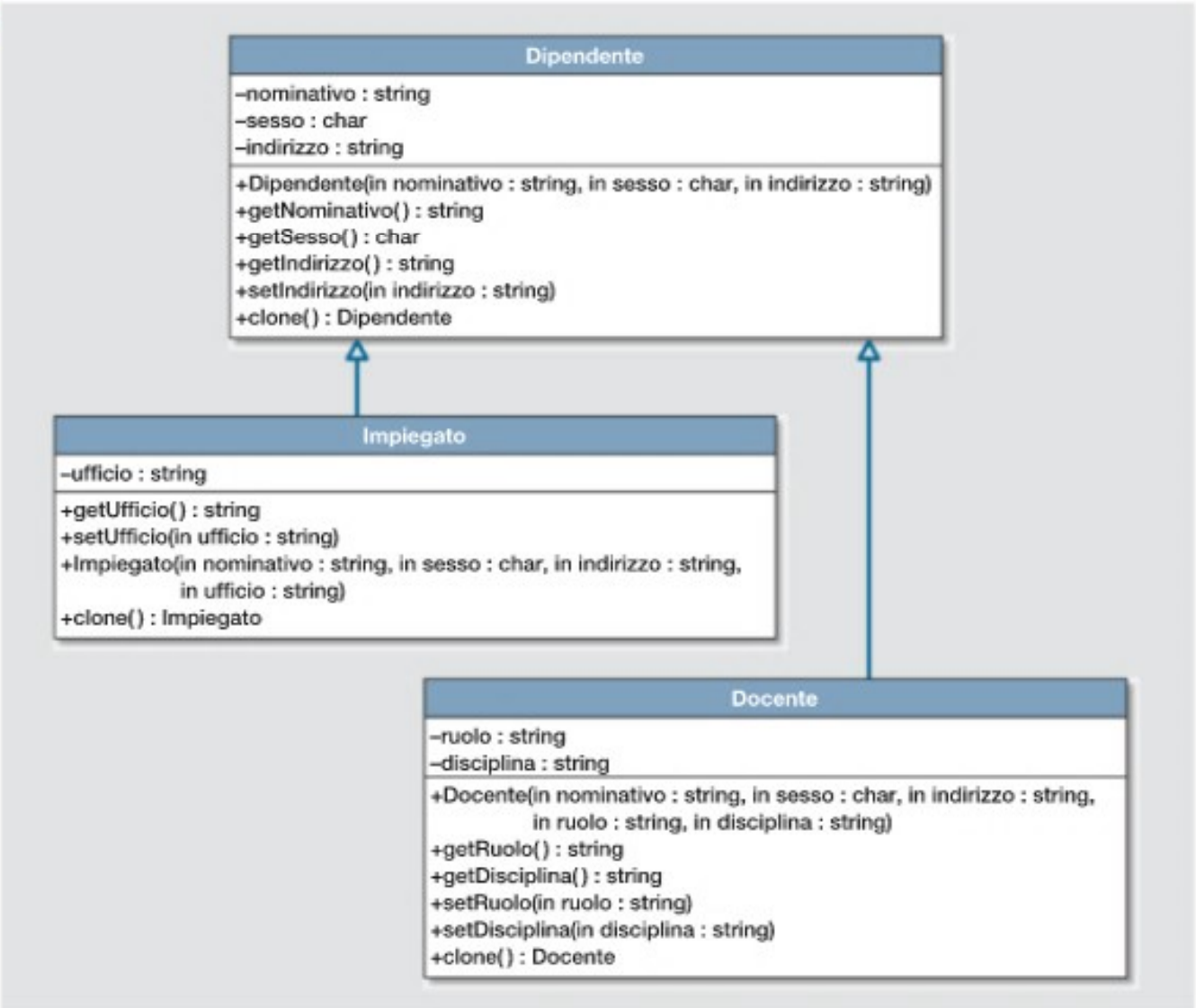


FIGURA 4

- aggiunta di un nuovo dipendente (impiegato o docente);
- eliminazione di un dipendente a partire dal nominativo;
- ricerca di un dipendente a partire dal nominativo;
- visualizzazione di tutti i dipendenti con indicazione dell'ufficio se impiegati, della disciplina di insegnamento se docenti.

La seguente classe Java implementa le funzionalità richieste memorizzando i dipendenti in un contenitore di tipo *ArrayList<T>* con capacità di 1000 elementi:

```
class TroppiDipendenti extends Exception {
}
class DipendenteSconosciuto extends Exception {
}

public class Dipendenti {
  public final static int NUMERO_MASSIMO_DIPENDENTI = 1000;
  private ArrayList<Dipendente> dipendenti; // contenitore

  // costruttore
  public Dipendenti() {
    dipendenti = new ArrayList<>(NUMERO_MASSIMO_DIPENDENTI);
  }

  // aggiunta dipendente
  public void aggiungiDipendente(Dipendente dipendente) throws TroppiDipendenti {
    if (dipendenti.size() >= NUMERO_MASSIMO_DIPENDENTI)
      throw new TroppiDipendenti();
    dipendenti.add(dipendente.clone()); // inserimento nel contenitore
  }

  // eliminazione dipendente
  public void eliminaDipendente(String nominativo) throws DipendenteSconosciuto {
    Iterator<Dipendente> i; // iteratore

    for (i=dipendenti.iterator(); i.hasNext();) {
      Dipendente dipendente = i.next();
      if (dipendente.getNominativo().equals(nominativo)) {
        i.remove(); // eliminazione dal contenitore
        return;
      }
    }
    throw new DipendenteSconosciuto();
  }

  // ricerca dipendente
  public Dipendente cercaDipendente(String nominativo)throws DipendenteSconosciuto {

    // ciclo for-each
    for (Dipendente dipendente : dipendenti) {
      if (dipendente.getNominativo().equals(nominativo))
        return dipendente.clone();
    }
    throw new DipendenteSconosciuto();
  }

  // elenco dipendenti
  public String elencoDipendenti() {
    String elenco = "";
```

```
    // ciclo for-each
    for (Dipendente dipendente : dipendenti) {
      elenco += dipendente.getNominativo() + " " + dipendente.getSesso() +
                "" + dipendente.getIndirizzo();
      if (dipendente instanceof Impiegato)
        elenco += " " + ((Impiegato)(dipendente)).getUfficio();
      if (dipendente instanceof Docente)
        elenco += " " + ((Docente)(dipendente)).getDisciplina();
      elenco += "\r\n"; // fine riga
    }
    return elenco;
  }
}
```

L'eliminazione di un dipendente comporta la sua ricerca sequenziale all'interno della collezione: a questo scopo è stato utilizzato un iteratore in modo esplicito. Il ciclo *for-each* che utilizza un iteratore implicito è stato invece impiegato per la ricerca di un dipendente e la costruzione dell'elenco. L'eventuale scelta della classe *LinkedList<T>* come contenitore avrebbe comportato modifiche minime al codice: l'eliminazione dell'impostazione della capacità del contenitore nell'invocazione del costruttore e del controllo sulla disponibilità di spazio effettuato nel metodo *aggiungiDipendente*.

2.2 Le classi contenitore che implementano l'interfaccia *Map<K, T>*

A differenza dell'interfaccia *Collection<T>*, l'interfaccia *Map<K, T>* è adottata da contenitori in cui gli elementi di tipo *T* sono riferiti – e talvolta ordinati – in base a una chiave identificativa di tipo *K* associata («mappata») all'elemento.

OSSERVAZIONE Molte entità del mondo reale hanno una chiave identificativa: il codice fiscale per le persone, la partita IVA per le aziende, la targa per gli autoveicoli, il codice a barre per i prodotti commerciali, ...

La chiave, oltre a identificare univocamente un'entità rappresentata da un oggetto da inserire come elemento in un contenitore, dovrebbe costituire l'informazione in base alla quale più frequentemente sono effettuate le ricerche nel contenitore stesso: generalmente le collezioni Java che implementano l'interfaccia *Map<K, T>* sono molto efficienti nell'effettuare la ricerca di un elemento in base alla chiave associata, ma la ricerca in base ad altri criteri è inevitabilmente sequenziale, e di conseguenza costosa, se il numero di elementi contenuti nella collezione è elevato.

ESEMPIO L'elenco dei numeri telefonici degli utenti di un operatore di telefonia mobile sarà memorizzato utilizzando come chiave il numero telefonico, anziché il nominativo, se lo scopo del contenitore è quello di identificare l'utente a partire dal numero di telefono.

L'interfaccia *Map<K, T>* definisce i metodi riportati nella TABELLA 5, che sono comuni a tutte le classi che la implementano (non sono stati riportati i metodi non specifici dell'interfaccia).

TABELLA 5

Firma del metodo	Descrizione della funzionalità
`void clear();`	Eliminazione di tutti gli elementi della mappa.
`boolean containsKey(Object k);`	Verifica della presenza della chiave *k* nella mappa.
`boolean containsValue(Object v);`	Verifica della presenza del valore *v* nella mappa.
`Set<Map.Entry<K,T>> entrySet();`	Restituisce una collezione costituita dalle associazioni chiave/elemento contenute nella mappa (*Map.Entry<K, T>* è la classe generica che definisce le associazioni chiave/elemento di una mappa).
`T get(Object k);`	Restituisce se esiste l'elemento associato alla chiave *k*.
`boolean isEmpty();`	Verifica se la mappa è vuota.
`Set<K> keySet();`	Restituisce una collezione costituita dalle chiavi degli elementi presenti nella mappa.
`T put(K k, T v);`	Inserimento dell'elemento con chiave *k* associata al valore *v* nella mappa.
`void putAll(Map<? extends K>, <? extends T>> m);`	Inserimento nella mappa di tutti gli elementi della mappa *m*.
`T remove (Object k);`	Rimozione dell'elemento avente chiave *k* dalla mappa.
`int size();`	Restituisce il numero di elementi presenti nella mappa.
`Collection<T> values();`	Restituisce una collezione costituita dai valori di tutti gli elementi presenti nella mappa.

Le due classi generiche concrete maggiormente usate per memorizzare elementi identificati da una chiave sono *TreeMap<K, T>* e *HashMap<K, T>*. La prima implementa un albero binario di ricerca bilanciato i cui nodi sono ordinati in funzione delle chiavi associate agli elementi, mentre la seconda implementa una tabella *hash* della quale nel costruttore è possibile definire la dimensione e il fattore di carico desiderato (il valore predefinito è 0,75): la tabella viene dinamicamente ridimensionata in modo da mantenere il fattore di carico impostato.

Le classi generiche *TreeMap<K, T>* e *HashMap<K, T>* hanno metodi specifici, ma implementano la stessa interfaccia ed espongono di conseguenza un insieme di metodi comuni; la scelta del tipo di contenitore corretto dipende quindi principalmente dall'uso che il programmatore intende farne e dalle prestazioni richieste; la TABELLA 6, permette di effettuare la scelta (*N* è il numero di elementi presenti nella collezione)[5].

5. M. Naftalin e P. Wadler, *Java Generics and Collections*, O'Reilly, 2006.

TABELLA 6

Classe	*containsKey*	*next (Iterator<T>)*	*get*
TreeMap<T>	Tempo proporzionale a log *N*.	Tempo proporzionale a log *N*.	Tempo proporzionale a log *N*.
HashMap<T>	Tempo costante.	Tempo costante.	Tempo proporzionale a *d/N* (*d* è la dimensione della tabella).

Considerando esclusivamente le prestazioni in tempo, il ricorso alla classe *HashMap<K, T>* è sicuramente preferibile: le prestazioni documentate nella precedente tabella sono però medie – a causa del fenomeno delle collisioni – e realistiche solo nel caso che il fattore di carico si mantenga inferiore al 75%, cosa che comporta un'occupazione di memoria di circa il 30% superiore allo spazio necessario per la memorizzazione degli oggetti che rappresentano gli elementi.
Inoltre non è possibile imporre un ordinamento agli elementi memorizzati in una tabella di tipo *HashMap<K, T>*, cosa che è invece possibile utilizzando un albero di tipo *TreeMap<K, T>*.

ESEMPIO

Con riferimento alla gerarchia di classi di FIGURA **5**, si intende progettare una classe Java che consenta la gestione dei libri di una grande biblioteca – che li classifica utilizzando un codice numerico identificativo – permettendo le seguenti operazioni di base:

- aggiunta di un nuovo libro o fumetto;
- eliminazione di un libro a partire dal codice;
- ricerca di un libro a partire dal codice;
- ricerca di un libro a partire dal titolo;
- ricerca di tutti i libri di un autore.

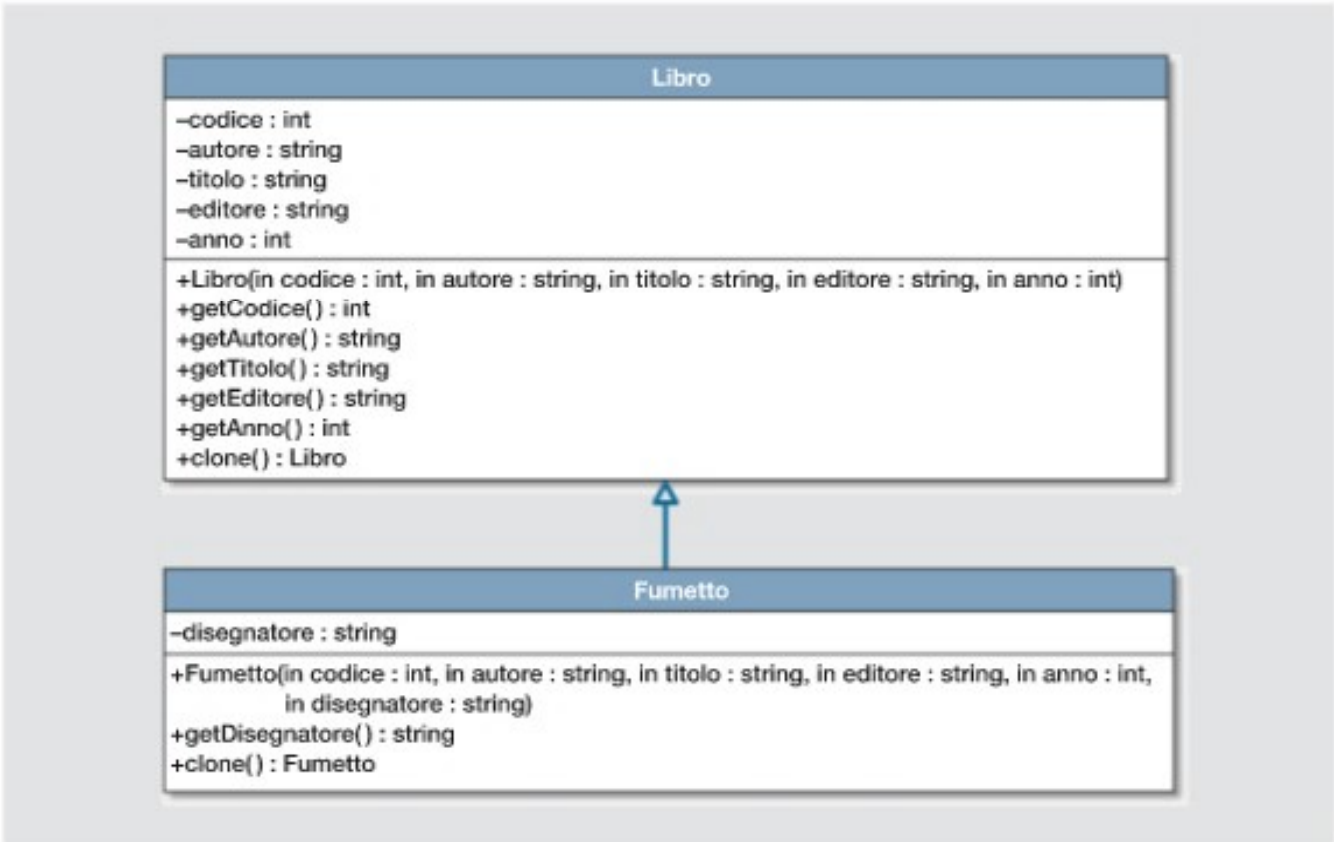


FIGURA **5**

La seguente classe Java implementa le funzionalità richieste memorizzando i libri in un contenitore di tipo *TreeMap<T>*:

```
class LibroEsistente extends Exception {
}
class LibroNonEsistente extends Exception {
}
```

```java
public class Biblioteca {
  private TreeMap<Integer, Libro> libri;

  // costruttore
  public Biblioteca() {
    libri = new TreeMap<Integer, Libro>();
  }

  // aggiunta libro
  public void aggiungiLibro(Libro libro) throws LibroEsistente {
  if (!libri.containsKey(libro.getCodice())) // verifica duplicato
    libri.put(libro.getCodice(), libro.clone()); // inserimento
  else
    throw new LibroEsistente();
  }

  // eliminazione libro
  public void eliminaLibro(int codice) throws LibroNonEsistente {
    if (libri.containsKey(codice)) // verifica esistenza chiave
      libri.remove(codice); // rimozione
    else
      throw new LibroNonEsistente();
  }

  // ricerca libro a partire dal codice
  public Libro trovaLibro(int codice) throws LibroNonEsistente {
    if (libri.containsKey(codice)) // verifica esistenza chiave
      return libri.get(codice).clone(); // accesso diretto elemento
    else
      throw new LibroNonEsistente();
  }

  // ricerca libro a partire dal titolo
  public Libro trovaLibro(String titolo) throws LibroNonEsistente {
    for (Libro libro : libri.values()) // scorrimento con iteratore
      if (libro.getTitolo().equals(titolo)) // verifica elemento
        return libro.clone();
    throw new LibroNonEsistente();
  }

  // ricerca libri a partire dall'autore
  public ArrayList<Libro> trovaLibri(String autore) {
    ArrayList<Libro> libri_autore = new ArrayList<Libro>(libri.size());
    for (Libro libro : libri.values()) // scorrimento con iteratore
      if (libro.getAutore().equals(autore)) // verifica elemento
        libri_autore.add(libro.clone());
    libri_autore.trimToSize();
    return libri_autore;
  }
}
```

Le operazioni basate sulla chiave di accesso agli elementi sono immediate, mentre le operazioni che non coinvolgono la chiave – come la ricerca di libri a partire dall'autore o dal titolo – richiedono una ricerca sequenziale effettuata mediante un ciclo *for-each* che utilizza implicitamente un iteratore: l'interfaccia *Map<K, T>* non prevede un iteratore, ma esso può essere ottenuto costruendo una collezione degli elementi della mappa mediante invocazione del metodo *values*. L'eventuale scelta della classe *HashMap<K, T>* come contenitore avrebbe comportato modifiche minime al codice, come la specificazione della dimensione iniziale della tabella.

2.3 Algoritmi generici: la classe di utilità *Collections*

La classe *Collections* del *package java.util* (da non confondere con l'interfaccia *Collection*) rende disponibili numerosi metodi statici che implementano varie funzioni di utilità su oggetti di tipo *Collection<T>* e/o *List<T>*.

La TABELLA 7 riporta i metodi più utilizzati dai programmatori trascurando in particolare i numerosi metodi *wrapper* della classe che restituiscono viste di una collezione in un formato diversamente strutturato e i metodi che prevedono la creazione di oggetti di tipo *Comparator*.

TABELLA 7

Firma del metodo	Descrizione della funzionalità
`public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> c);`	Restituisce il minimo tra gli elementi di una collezione *c* che implementano l'interfaccia *Comparable<T>*.
`public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> c);`	Restituisce il massimo tra gli elementi di una collezione *c* che implementano l'interfaccia *Comparable<T>*.
`public static <T extends Comparable<? super T>> void sort(List<T> l);`	Ordina gli elementi della lista *l* che implementano l'interfaccia *Comparable<T>*.
`public static <T> int binarySearch(List<T extends Comparable<? super T>> l, T e);`	Ricerca l'elemento *e* nella lista *l* mediante un algoritmo di ricerca binaria che presuppone ordinati gli elementi della lista stessa che implementano l'interfaccia *Comparable<T>*; restituisce l'indice di posizione dell'elemento *e* nella lista *l*: nel caso che esso non esista, restituisce un valore negativo che rappresenta l'indice della posizione in cui è possibile inserire l'elemento *e* mantenendo l'ordinamento di *l*.
`public static <T> void copy(List<? super T> d, List<? extends T> s);`	Copia gli elementi della lista *s* nella lista *d*.

OSSERVAZIONE La classe *Collections* non è generica, ma i suoi metodi sì: le loro firme sono precedute da un parametro di tipo successivamente utilizzato per la definizione dei parametri o del risultato.

ESEMPIO Disponendo di una classe *Persona* che implementa l'interfaccia generica *Comparable<T>* definendo il metodo *compareTo* – che può, per esempio, confrontare due diverse persone in base all'ordinamento alfabetico del cognome – la seguente dichiarazione istanzia un contenitore di persone di tipo *ArrayList<T>*

```
public ArrayList<Persona> persone = new ArrayList<>(1000);
```

che può essere ordinato invocando il metodo statico *sort* della classe *Collections* (l'ordinamento avviene sulla base dell'ordinamento «naturale» determinato dal metodo *compareTo* definito dal programmatore e utilizzato dall'algoritmo di ordinamento):

```
Collections.sort(persone);
```

I metodi che seguono consentono una gestione del contenitore che ne mantiene l'ordinamento:

```
public void inserisciPersona(Persona p) throws PersonaEsistente {
  int i = Collections.binarySearch(persone, p);
  if (i >= 0)
    throw new PersonaEsistente();
  persone.add(-i, p.clone());
}
```

```
public Persona cercaPersona(Persona p) throws PersonaInesistente {
  int i = Collections.binarySearch(persone, p);
  if (i < 0)
    throw new PersonaInesistente();
  return persone.get(i).clone();
}

public void eliminaPersona(Persona p) throws PersonaInesistente {
  int i = Collections.binarySearch(persone, p);
  if (i < 0)
    throw new PersonaInesistente();
  persone.remove(i);
  Collections.sort(persone);
}
```

I CONCETTI CHIAVE

TIPI PARAMETRICI. Un tipo parametrico del linguaggio Java è un identificatore di tipo che nella definizione di una classe – che in questo caso prende il nome di *classe generica* – non viene esplicitato: nella classe generica il tipo parametrico è rappresentato da un identificatore che deve essere istanziato al momento della dichiarazione di un riferimento alla classe.

CLASSE GENERICA. È una classe che nella sua definizione elenca i tipi parametrici utilizzati dopo il nome della classe tra i simboli «<» e «>»; la dichiarazione di un riferimento avente come tipo una classe generica elenca i tipi effettivamente istanziati dopo il nome della classe tra i simboli «<» e «>»: per istanziare tipi primitivi è necessario impiegare la classe *wrapper* associata. L'utilità delle classi generiche è evidente nel caso in cui si devono progettare classi aventi la funzione di contenitori di oggetti che possono essere di vario tipo ed è pertanto necessario parametrizzarli (per esempio una classe generica *Coda<T>* consente di dichiarare una coda relativa a oggetti di un qualsiasi tipo *T*). Prima della versione 5, il linguaggio di programmazione Java non disponeva dei tipi generici: per definire classi contenitore di tipo non vincolato si ricorreva ad attributi e parametri di tipo *Object* (con la conseguente necessità di effettuare le opportune operazioni di *casting*) che offrono la possibilità di riferire un qualsiasi oggetto Java.

TIPO *RAW*. È una classe generica utilizzata senza istanziare il tipo parametrico: in questo caso il compilatore non effettua controlli relativi al tipo parametrico, che risulta in pratica assimilabile al tipo *Object*. I tipi *raw* sono stati introdotti per facilitare l'integrazione di codice Java scritto prima della versione 5 del linguaggio, che ha introdotto la classi generiche, e non dovrebbero essere utilizzati dai programmatori perché comportano lo spostamento del controllo della correttezza dei tipi dal momento della compilazione alla fase di esecuzione.

VINCOLI DEI TIPI PARAMETRICI. Non sempre il progettista di una classe generica intende consentire che la classe sia istanziata con tipi del tutto generali: è possibile vincolare il tipo parametrico a essere sottotipo di una classe/interfaccia, o super-tipo di una classe utilizzando le clausole *<T extends S>* (il tipo *T* deve derivare dalla classe *S* o implementare l'interfaccia *S*), *<T extends C & I>* (il tipo *T* deve derivare dalla classe *C* e implementare l'interfaccia *I*), *<T super C>* (il tipo *T* deve essere super-tipo dalla classe *C*).

TIPI PARAMETRICI JOLLY. Java prevede l'uso del carattere jolly «?» (*wildcard*) che permette di definire vincoli per i tipi parametrici anche quando si intendono riferire intere famiglie di tipi piuttosto che un singolo tipo. Le clausole previste in questo caso sono: *<?>* (tipo senza limiti), *<? extends S>* (tipo con limite superiore: deve essere sottotipo di *S*), *<? super S>* (tipo con limite inferiore: deve essere super-tipo di *S*). Il tipo jolly con limite superiore (*<? extends S>*) viene spesso impiegato per i metodi che estraggono/copiano (*get*) i valori *da* un oggetto istanza di una classe generica, mentre il tipo jolly con limite inferiore (*<? super S>*) è spesso utilizzato per i metodi che inseriscono/copiano (*set*) i valori *in* un oggetto istanza di una classe generica.

COLLEZIONI. Il *Collection framework* del linguaggio Java è un insieme di interfacce e di classi definite nel *package java.util* aventi lo scopo di implementare contenitori di oggetti (liste, code, alberi, tabelle, ...) in modo efficace ed efficiente per il programmatore. Le interfacce generiche fondamentali del *framework* definiscono in modo astratto i tipi di contenitori resi disponibili: *List<T>* (*Arraylist<T>*, *LinkedList<T>*), *Queue<T>* (*PriorityQueue<T>*), *Set<T>* (*HashSet<T>*, *TreeSet<T>*), *Map<T>* (*HashMap<T>*, *TreeMap<T>*). Tutte le interfacce che derivano dall'interfaccia *Collection<T>* ereditano l'interfaccia *Iterable<T>*, che richiede l'implementazione di un metodo *iterator* che restituisce un iteratore che permette di scorrere gli elementi della collezione: è quindi possibile impiegare il ciclo *for-each* che utilizza l'iteratore in modo implicito per scorrere gli elementi contenuti nella classe; le classi che implementano l'interfaccia *Map<K, T>* non implementano l'interfaccia *Iterable*, ma dispongono del metodo *values*, che estrae dal contenitore una collezione dotata di iteratore.

INTERFACCIA *COMPARABLE<T>*. Molte classi generiche utilizzabili per realizzare un contenitore impongono agli elementi di tipo parametrico *T* e/o alle chiavi di tipo parametrico *K* un ordinamento: a questo scopo, nel caso di classi definite dal programmatore, è buona norma che esse implementino l'interfaccia *Comparable<T>*, definita nel *package java.lang* che richiede l'implementazione del metodo *int compareTo(T object)* che deve restituire un valore intero negativo, nullo o positivo a seconda che l'oggetto su cui viene invocato sia minore, uguale o maggiore dell'oggetto fornito come parametro.

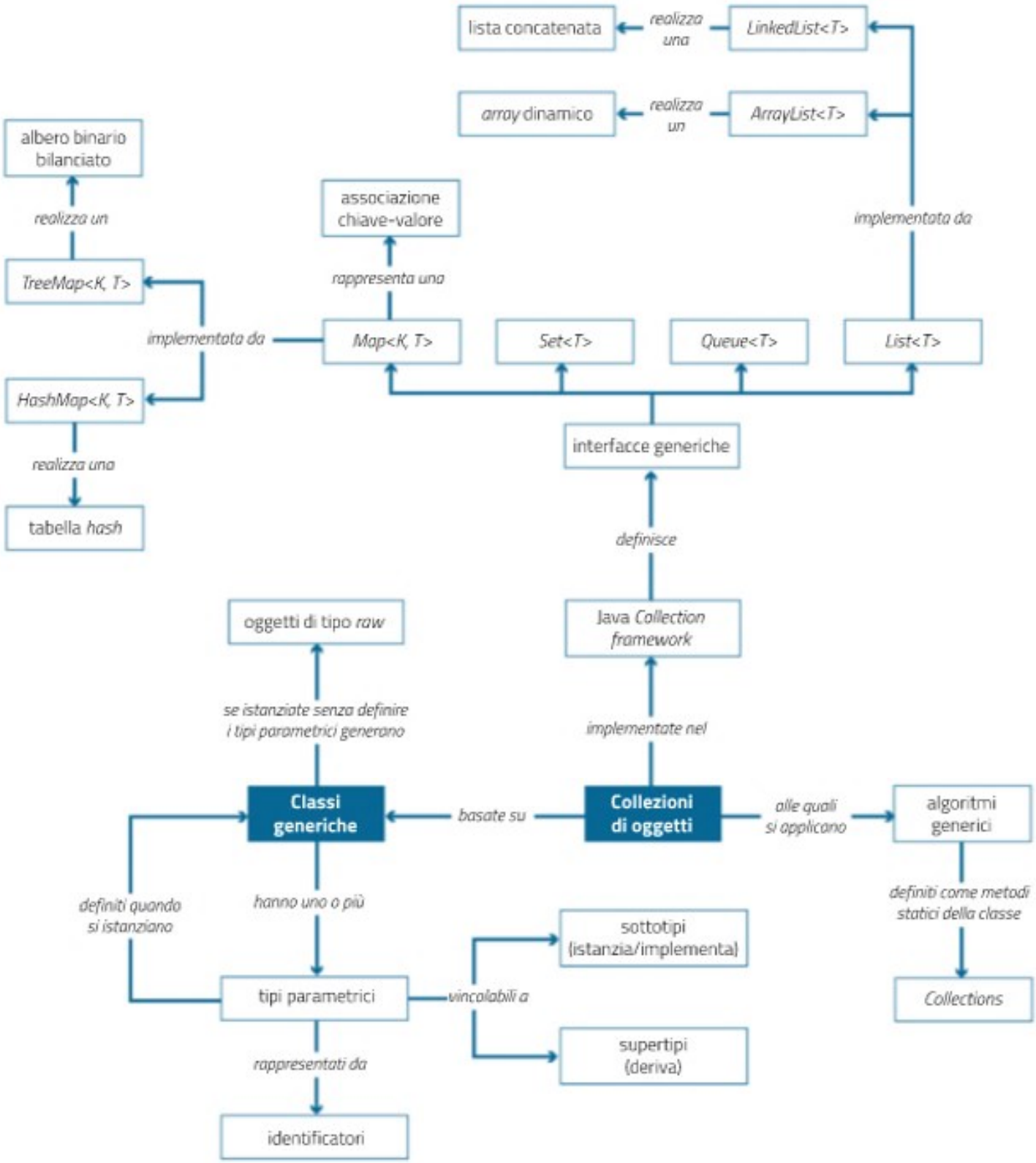
ARRAYLIST<T>* E *LINKEDLIST<T>. Sono le classi concrete generiche principali che implementano le interfacce *Collection<T>* e *List<T>*: la prima – in analogia a un *array* – prevede una collezione di un numero massimo di elementi che può essere impostato come parametro del costruttore e successivamente modificato invocando il metodo *ensureCapacity*; la seconda è – essendo una vera e propria lista concatenata – indefinitamente estensibile. *ArrayList<T>* è la scelta corretta se l'uso del contenitore prevede prevalentemente accessi in base alla posizione e un limitato numero di rimozioni;

viceversa, se si hanno frequenti rimozioni e un accesso prevalentemente sequenziale, allora è preferibile *LinkedList<T>* che, oltre ai metodi definiti dall'interfaccia *List<T>*, implementa i tradizionali metodi di accesso a una lista concatenata per l'inserimento/eliminazione di elementi in testa e in coda.

TREEMAP<K, T>* E *HASHMAP<K, T>. Sono le due classi generiche concrete maggiormente usate per memorizzare elementi identificati da una chiave: la prima implementa un albero binario di ricerca bilanciato i cui nodi sono ordinati in funzione delle chiavi associate agli elementi, mentre la seconda implementa una tabella *hash* della quale nel costruttore è possibile definire la dimensione e il fattore di carico desiderato (il valore predefinito è 0,75): la tabella viene dinamicamente ridimensionata in modo da mantenere il fattore di carico impostato. Non è possibile imporre un ordinamento agli elementi memorizzati in una tabella di tipo *HashMap<K, T>*, caratteristica che è invece predefinita utilizzando un albero di tipo *TreeMap<K, T>*.

CLASSE *COLLECTIONS*. La classe di utilità *Collections* del *package java.util* (da non confondere con l'interfaccia *Collection*) rende disponibili numerosi metodi statici che implementano varie funzioni di utilità su oggetti di tipo *Collection<T>* e/o *List<T>* tra i quali: restituzione del minimo/massimo degli elementi di una collezione, ordinamento degli elementi di una lista, ricerca binaria di un elemento in una lista, copia degli elementi tra due liste, ...

RIPASSA CON LA MAPPA



SPIEGA IN UNA PAGINA

1 **Descrivere, utilizzando anche esempi di codice, l'uso dei tipi parametrici e delle relative classi generiche nel linguaggio Java.**

2 **Illustrare, anche mediante esempi di codice, l'uso del Java *Collection framework* per la gestione di collezioni di oggetti nel linguaggio Java.**

QUESITI A RISPOSTA APERTA

3 **Che cosa significa vincolare superiormente o inferiormente un tipo generico?**

4 **Quali collezioni del Java *Collection framework* implementano rispettivamente un albero binario di ricerca bilanciato e una tabella *hash*?**

5 **Per quali applicazioni è preferibile utilizzare la classe generica *ArrayList<T>* del Java *Collection framework* piuttosto che la classe generica *LinkedList<T>*?**

TEST A RISPOSTA MULTIPLA

6 **Un tipo parametrico del linguaggio Java è l'identificatore di tipo:**

A di un qualsiasi parametro passato a un qualsiasi metodo di una classe.

B che nella definizione di una classe non viene esplicitato, ma che deve essere istanziato al momento della dichiarazione di un riferimento alla classe.

C da specificare all'atto della definizione di una gerarchia di classi che permette di parametrizzare il tipo di ereditarietà.

D alternativo da utilizzare per la compilazione di una classe in caso di errori causati dal tipo predefinito.

7 **Nel linguaggio Java una classe generica è:**

A una classe qualsiasi di cui non sia stato specificato il nome.

B una classe qualsiasi di cui non sia stato specificato il tipo.

C una classe che nella sua definizione prevede tipi parametrici.

D la stessa cosa di una classe astratta.

8 **Un'alternativa ai tipi generici per definire classi contenitore di tipo non vincolato è utilizzare:**

A attributi e parametri di tipo *String*.

B classi *wrapper*.

C attributi e parametri di tipo *Object*.

D Non ci sono alternative praticabili al ricorso ai tipi generici per questa applicazione.

9 **Un tipo *raw* corrisponde:**

A a una classe generica utilizzata senza istanziarne i tipi parametrici.

B a una classe parametrica utilizzata per istanziare un tipo generico.

C a un qualsiasi tipo parametrico.

D al tipo *Object*.

10 **Indicare quali delle seguenti affermazioni relative alle classi generiche sono vere.**

A È possibile derivare classi generiche da altre classi generiche.

B Le classi generiche non hanno tipi parametrici.

C Le classi generiche hanno uno o più tipi parametrici.

D È possibile derivare classi generiche da classi non generiche.

11 **Dovendo definire un metodo statico per calcolare la somma degli elementi di una lista generica fornita come parametro, è necessario:**

A disporre di un iteratore per scorrere la lista.

B che il tipo parametrico su cui la lista è istanziata sia un super-tipo della classe *Number*.

C che il tipo parametrico su cui la lista è istanziata sia derivato dalla classe *Number*.

D che il tipo parametrico su cui la lista è istanziata sia un tipo primitivo.

12 **I tipi parametrici jolly servono per:**

A definire vincoli per i tipi parametrici quando si intende riferire una famiglia di tipi piuttosto che un singolo tipo.

B definire vincoli per i tipi parametrici quando si intende riferire uno specifico tipo piuttosto che una famiglia di tipi.

C specificare come tipi parametrici classi definite in un diverso linguaggio di programmazione.

D definire vincoli per i tipi parametrici, ma esclusivamente per gli attributi di una classe e non per i parametri dei metodi.

13 **Il *Collection framework* del linguaggio Java è:**

A un insieme di classi concrete definite nel *package java.util* aventi lo scopo di implementare contenitori di oggetti (liste, code, alberi, tabelle, ...).

B un insieme di classi astratte definite nel *package java.util* aventi lo scopo di implementare contenitori di oggetti (liste, code, alberi, tabelle, ...).

C un insieme di interfacce e di classi definite nel *package java.util* aventi lo scopo di implementare contenitori di oggetti (liste, code, alberi, tabelle, ...).

D Il linguaggio Java non ha un *Collection framework*.

14 **L'interfaccia generica *Comparable<T>* è funzionale all'implementazione di:**

A ordinamenti su oggetti di classi definite dal programmatore.

B strutture ad albero binario su oggetti di classi generiche.

C indirizzamenti *hash* su oggetti di classi astratte.

D Non esiste questa interfaccia generica tra i *package* di utilità del linguaggio Java.

15 **Per l'implementazione dei tradizionali metodi di accesso (inserimento/estrazione in testa e/o coda, ...) a una lista di oggetti è opportuno utilizzare una classe generica del tipo:**

A *TreeMap<K, T>*.

B *LinkedList<T>*.

C *HashMap<K, T>*.

D *ArrayList<T>*.

16 **Indicare per quali classi generiche che implementano l'interfaccia generica *Collection<T>* è possibile invocare il metodo statico *sort* della classe *Collections*:**

A *TreeMap<K, T>*.

B *LinkedList<T>*.

C *HashMap<K, T>*.

D *ArrayList<T>*.

17 **Indicare quali delle seguenti affermazioni relative alla classe *Collections* sono vere.**

A Equivale all'interfaccia *Collection*.

B È una classe generica.

C Ha metodi che permettono di determinare il minimo e/o massimo tra gli elementi di una collezione.

D Ha metodi che permettono di ordinare e di effettuare ricerche binarie su contenitori di tipo lista.

PROBLEMI

18 **Codificare in linguaggio Java una classe generica *Stack* che implementi una pila di elementi di tipo parametrico. Scrivere un metodo *main* che verifichi le funzionalità della classe istanziandola per vari tipi di elementi.**

19 **Si deve realizzare una rubrica per la gestione dei numeri telefonici dei contatti. Dopo aver progettato mediante un diagramma UML una classe che modelli l'entità «contatto», implementare in Java una classe che, utilizzando un contenitore del *Collection framework* del linguaggio, consenta la gestione completa della rubrica e, in particolare, le seguenti operazioni:**

- **aggiunta di un nuovo contatto;**
- **salvataggio su file dell'intera rubrica;**
- **caricamento da file dell'intera rubrica;**
- **visualizzazione delle informazioni di contatto a partire dal numero di telefono;**
- **eliminazione di un contatto specificando il numero di telefono;**
- **visualizzazione delle informazioni di contatto a partire dall'indirizzo e-mail;**
- **eliminazione di un contatto specificando l'indirizzo e-mail.**

20 **A un ufficio postale accedono clienti che scelgono di usufruire di uno dei seguenti tipi di servizio:**

- **servizio di spedizione (*S*);**
- **servizio di ricezione (*R*);**
- **servizio finanziario (*F*).**

I singoli sportelli dell'ufficio postale sono specializzati per effettuare uno specifico servizio e i clienti sono serviti nell'ordine in cui sono arrivati in base al servizio richiesto (può accadere quindi

che un cliente *X* che richiede il servizio *S* sia servito prima del cliente *Y* che richiede un diverso servizio anche se *Y* è giunto prima di *X*).

A. Codificare una classe Java che rappresenti un cliente identificato da un codice numerico progressivo generato in fase di creazione di un oggetto; la classe deve implementare l'interfaccia *Comparable<T>* garantendo l'ordinamento naturale del codice numerico generato.

B. Codificare una classe Java che modelli il funzionamento dell'ufficio postale utilizzando tre istanze della classe *PriorityQueue<T>* (una per ciascuno dei servizi) consentendo di simulare le seguenti attività:
   - arrivo di un nuovo cliente che richiede uno specifico servizio;
   - chiamata di un cliente in attesa per un servizio da parte di uno sportello;
   - visualizzazione del numero dei clienti in attesa per ciascuno dei tre servizi.

C. Modificare le classi in modo da gestire con priorità e per il solo servizio *F* i clienti in possesso di una carta *postepay*.

CASI PRATICI

21 A SCUOLA DI LAVORO Una biblioteca intende gestire con un'applicazione Java i libri posseduti (identificati dal codice ISBN) e i soci che accedono al prestito (identificati dal codice fiscale). I prestiti in corso devono essere rappresentati mediante un elenco di coppie (codice-libro, codice-socio): la singola coppia deve essere rimossa al momento della restituzione.
Il capo-progetto ha deciso l'uso dei seguenti contenitori Java:

Libri	*HashMap*
Soci	*TreeMap*
Prestiti	*ArrayList*

L'applicazione deve consentire:
- la ricerca dei dati di un libro a partire dal proprio codice ISBN;
- la ricerca dei dati di un socio a partire dal proprio codice fiscale;
- la verifica del prestito di un libro con indicazione dei dati del socio che lo ha preso;
- la memorizzazione di un nuovo prestito;
- l'aggiornamento dei prestiti quando avviene una restituzione;
- il salvataggio dei dati (libri, soci, prestiti) su file;
- il recupero dei dati (libri, soci, prestiti) da file.

22 A SCUOLA DI LAVORO Un corriere internazionale che serve esclusivamente clienti abbonati è organizzato in modo che al momento in cui viene ricevuta una spedizione identificata dal codice assegnato al momento in cui è stata consegnata, vengono visualizzate sul computer dell'incaricato le informazioni (denominazione, indirizzo, città, nazione, telefono, ...) del mittente e del destinatario della spedizione.
Dopo aver completato la seguente tabella

Struttura dati	Implementazione Java della collezione	Motivazione
Clienti		
Spedizioni		

implementare in linguaggio Java le classi indicate nella tabella codificando in particolare i metodi che consentono di aggiungere elementi alla collezione e di ricercare uno specifico elemento della collezione generando le eventuali eccezioni necessarie. Implementare inoltre una classe con un metodo *ricezione* che, a partire dal codice della spedizione, visualizzi le informazioni per l'incaricato gestendo le eventuali eccezioni.

23 Il call-center di una grande società di servizi è organizzato in modo tale che al momento in cui viene ricevuta una telefonata identificata dal numero chiamante vengono visualizzate sul computer del dipendente incaricato della risposta le seguenti informazioni:
- codice del cliente;
- cognome e nome del cliente;
- indirizzo e città del cliente;
- data e ora dell'ultima telefonata effettuata;
- codice, cognome e nome del dipendente che ha risposto all'ultima telefonata effettuata.

Dopo aver completato la seguente tabella

Struttura dati	Implementazione Java della collezione	Motivazione
Clienti		
Dipendenti		
Telefonate		

implementare in linguaggio Java le classi indicate nella tabella codificando in particolare i metodi che consentono di aggiungere elementi alla collezione e di ricercare uno specifico elemento della collezione generando le eventuali eccezioni necessarie. Implementare inoltre una classe con un metodo *chiamata* che, a partire dal numero telefonico chiamante, visualizzi le informazioni per il dipendente incaricato della risposta gestendo le eventuali eccezioni.

24 A SCUOLA DI LAVORO Si intende sviluppare una semplice applicazione Java che consenta di gestire gli ingressi e le uscite di materiale da un magazzino. A questo scopo devono essere progettate e sviluppate le classi che implementano una collezione di «operazioni» (ingressi o uscite a cui viene automaticamente assegnato un codice di operazione) e ne realizzano la gestione eseguendo le seguenti azioni:

- nuovo ingresso specificando importo, quantità e descrizione;
- nuova uscita specificando importo, quantità e descrizione;
- eliminazione di un'operazione precedentemente inserita;
- visualizzazione dei dati relativi a un'operazione di cui è nota la descrizione;
- visualizzazione della quantità totale e del valore totale della merce immagazzinata;
- salvataggio su file della lista delle operazioni;
- caricamento da file della lista delle operazioni;
- visualizzazione di tutte le operazioni effettuate.

25 Si intende sviluppare una semplice applicazione Java che consenta di gestire gli incassi e le spese personali. A questo scopo devono essere progettate e sviluppate le classi che implementano una collezione di «operazioni» (incassi o spese a cui viene automaticamente assegnato un codice di operazione) e ne realizzano la gestione eseguendo le seguenti azioni:

- nuovo incasso specificando importo e motivo;
- nuova spesa specificando importo e motivo;
- eliminazione di un'operazione precedentemente inserita;
- visualizzazione dei dati relativi a un'operazione di cui è nota la descrizione;
- visualizzazione dei dati relativi a un'operazione di cui è noto il codice;
- salvataggio su file della lista delle operazioni;
- caricamento da file della lista delle operazioni;
- visualizzazione di tutte le operazioni effettuate.

PROGETTI

26 Le autovetture da consegnare ai concessionari per la vendita non sono ancora dotate di targa e sono di conseguenza identificate dal solo numero di telaio e caratterizzate dalla marca, dal modello e dal peso a vuoto; per le automobili con alimentazione a gas è inoltre necessario conoscere il volume dei serbatoi. Allo scopo di sviluppare l'applicazione software per la gestione di una grande area di smistamento delle vetture con postazioni identificate da una stringa alfanumerica (per esempio: «AREA NORD 9999») sono richiesti:

- un diagramma UML che modelli la singola autovettura;
- la/e classe/i Java che implementa/no il modello UML;
- l'individuazione motivata di una collezione Java che consenta di modellare l'area di smistamento;
- il codice di una classe Java che implementi la gestione dell'area di smistamento realizzando le seguenti operazioni:
  a) ingresso di una nuova autovettura nell'area specificandone la postazione,
  b) uscita dall'area di un'autovettura di cui è noto il numero di telaio,
  c) ricerca dei dati relativi a un'autovettura specificandone la postazione,
  d) calcolo del peso totale di un gruppo di vetture i cui numeri di telaio sono forniti come vettore,
  e) calcolo del volume totale dei serbatoi di gas di un gruppo di vetture le cui postazioni sono fornite come vettore;
- il salvataggio su file e il relativo caricamento da file dell'intero parco vetture presente nell'area di smistamento.

All'area di smistamento le autovetture arrivano e partono trasportate da camion: dopo aver individuato una struttura dati che consenta di modellare un camion per il trasporto di autovetture, implementare la relativa classe Java prevedendo metodi per il caricamento e lo scaricamento di una singola autovettura. Derivare dalla classe che modella l'area di smistamento una nuova

classe che preveda i metodi *scaricaCamion* e *caricaCamion* che implementano le operazioni di spostamento delle autovetture dal camion all'area e viceversa e i cui parametri sono:

- il camion;
- un elenco di postazioni che identifica le autovetture da caricare, o le posizioni in cui collocare le autovetture scaricate.

27 I container di una grande linea di spedizioni internazionali sono identificati da un codice numerico e caratterizzati da stazza e carico; per i container refrigerati è inoltre necessario conoscere la temperatura di mantenimento del contenuto. Allo scopo di sviluppare l'applicazione software per la gestione di un grande deposito con postazioni numerate per i container della linea di spedizioni sono richiesti:

- un diagramma UML che modelli il singolo container;
- la/e classe/i Java che implementa/no il modello UML;
- l'individuazione motivata di una collezione Java che consenta di modellare il deposito;
- il codice di una classe Java che implementi la gestione del deposito gestendo le seguenti operazioni:
  a) ingresso di un nuovo container al deposito specificando la postazione,
  b) uscita di un container di cui è nota la postazione dal deposito,
  c) ricerca dei dati relativi a un container di cui è noto il codice,
  d) calcolo del carico totale di tutti i container presenti nel deposito,
  e) calcolo del carico totale dei soli container refrigerati presenti nel deposito;
- il salvataggio su file dello stato del deposito;
- il caricamento da file dello stato del deposito.

Il deposito è accessibile a treni merci capaci di caricare molti container. Individuare una collezione Java che consenta di modellare un treno e implementare la relativa classe Java prevedendo metodi per il caricamento e lo scaricamento di un singolo container. Derivare dalla classe che modella il deposito una nuova classe che preveda i metodi *scaricaTreno* e *caricaTreno* che implementano le operazioni di spostamento dei container dal treno al deposito e viceversa e i cui parametri sono:

- il treno;
- un elenco di codici numerici che identificano i container da caricare o scaricare.

arguably
probabilmente

backed
sostenuto
appoggiato

to belong
appartenere

by the way
a proposito

misnomer
improprio
designazione erronea

motto
motto, massima morale o proverbiale

perhaps
forse

slightly
leggermente

synergistic
sinergico

trivial
banale

Part I Generics

[...]

Chapter 1 Introduction

Generics and collections work well with a number of other new features introduced in the latest versions of Java, including boxing and unboxing, a new form of loop, and functions that accept a variable number of arguments. We begin with an example that illustrates all of these. As we shall see, combining them is synergistic: the whole is greater than the sum of its parts.

Taking that as our motto, let's do something simple with sums: put three numbers into a list and add them together. Here is how to do it in Java with generics:

```
List<Integer> ints = Arrays.asList(1,2,3);
int s = 0;
for (int n : ints) { s += n; }
assert s == 6;
```

You can probably read this code without much explanation, but let's touch on the key features. The interface List and the class Arrays are part of the Collections Framework (both are found in the package java.util). The type List is now generic; you write List<E> to indicate a list with elements of type E. Here we write List<Integer> to indicate that the elements of the list belong to the class Integer, the wrapper class that corresponds to the primitive type int. Boxing and unboxing operations, used to convert from the primitive type to the wrapper class, are automatically inserted. The static method asList takes any number of arguments, places them into an array, and returns a new list backed by the array. The new loop form, foreach, is used to bind a variable successively to each element of the list, and the loop body adds these into the sum. The assertion statement (introduced in Java 1.4), is used to check that the sum is correct; when assertions are enabled, it throws an error if the condition does not evaluate to True.

Here is how the same code looks in Java before generics:

```
List ints = Arrays.asList( new Integer[] {
  new Integer(1), new Integer(2), new Integer(3)
} );
int s = 0;
for (Iterator it = ints.iterator(); it.hasNext(); ) {
  int n = ((Integer)it.next()).intValue();
  s += n;
}
assert s == 6;
```

Reading this code is not quite so easy. Without generics, there is no way to indicate in the type declaration what kind of elements you intend to store in the list, so instead of writing List<Integer>, you write List. Now it is the coder rather than the compiler who is responsible for remembering the type of the list elements, so you must write the cast to (Integer) when extracting elements from the list. Without boxing and unboxing, you must explicitly allocate each object belonging to the wrapper class Integer and use the intValue method to extract the corresponding primitive int. Without functions that accept a variable number of arguments, you must explicitly allocate an array to pass to the asList method. Without the new form of loop, you must explicitly declare an iterator and advance it through the list.

By the way, here is how to do the same thing with an array in Java before generics:

```
int[] ints = new int[] { 1,2,3 };
int s = 0;
for (int i = 0; i < ints.size; i++) { s += ints[i]; }
assert s == 6;
```

This is slightly longer than the corresponding code that uses generics and collections, is arguably a bit less readable, and is certainly less flexible. Collections let you easily grow or shrink the size of the collection, or switch to a different representation when appropriate, such as a linked list or hash table or ordered tree. The introduction of generics, boxing and unboxing, foreach loops, and varargs in Java marks the first time that using collections is just as simple, perhaps even simpler, than using arrays.

1.1 Generics

An interface or class may be declared to take one or more type parameters, which are written in angle brackets and should be supplied when you declare a variable belonging to the interface or class or when you create a new instance of a class.

We saw one example in the previous section. Here is another:

```
List<String> words = new ArrayList<String>();
words.add("Hello");
words.add("world!");
String s = words.get(0)+ words.get(1);
assert s.equals("Hello world!");
```

In the Collections Framework, class ArrayList<E> implements interface List<E>. This trivial code fragment declares the variable words to contain a list of strings, creates an instance of an ArrayList, adds two strings to the list, and gets them out again.

In Java before generics, the same code would be written as follows:

```
List words = new ArrayList();
words.add("Hello");
words.add("world!");
String s = ((String)words.get(0))+((String)words.get(1))
assert s.equals("Hello world!");
```

Without generics, the type parameters are omitted, but you must explicitly cast whenever an element is extracted from the list.

In fact, the bytecode compiled from the two sources above will be identical. We say that generics are implemented by erasure because the types List<Integer>, List<String>, and List<List<String>> are all represented at run-time by the same type, List. We also use erasure to describe the process that converts the first program to the second. The term erasure is a slight misnomer, since the process erases type parameters but adds casts.

[...]

[Maurice Naftalin and Philip Wadler, *Java Generics and Collections*, O'Reilly, 2006]

QUESTIONS

a **What are Java generics?**

b **Why are generics described as being implemented by erasure?**

c **What is the correct syntax for declaring one or more type parameters for interface or class?**

d **What does the expression *List<Integer>* indicate in the following statement?**

```
List<Integer> ints = Arrays.asList(1,2,3);
```

A7

La programmazione funzionale in Java

La programmazione funzionale è un paradigma di programmazione in cui la computazione viene espressa come l'applicazione di una serie di funzioni. Si tratta di uno stile di programmazione dichiarativo, in quanto il programmatore si concentra sulla descrizione di «cosa calcolare», e non sul «come farlo». La programmazione funzionale pone infatti l'accento sulla descrizione delle funzioni da calcolare, e non sulla sequenza di istruzioni che sono necessarie a implementare il calcolo di tali funzioni. Il flusso di esecuzione di un programma funzionale viene poi strutturato applicando e componendo le funzioni dichiarate.

Il seguente codice Java, dopo aver inizializzato una lista contenente i nomi di tre specie animali, visualizza tali nomi sullo schermo.

```
public static void main(String[] args) {
  List<String> animali = Arrays.asList("Cane", "Gatto", "Topo");

  for (String animale : animali) {
    System.out.println(animale);
  }
}
```

Il codice mostrato sopra è puramente imperativo, in quanto richiede di descrivere come iterare sulla lista animali (ovvero attraverso un ciclo **for**), e cosa fare a ogni iterazione (ovvero visualizzare il contenuto della variabile *animale*). La stessa computazione può essere espressa in modo più semplice e conciso attraverso la programmazione funzionale. Quest'ultima, infatti, richiede solamente di descrivere la funzione da applicare a ogni elemento della lista.

```
public static void main(String[] args) {
  List<String> animali = Arrays.asList("Cane", "Gatto", "Topo");

  animali.forEach(animale -> System.out.println(animale));
}
```

La programmazione funzionale intende le funzioni nel senso matematico del termine. L'output restituito da una funzione dipende sempre e solo dagli argomenti passati in input a tale funzione. In altre parole, dato uno stesso input x, una funzione f restituirà sempre lo stesso valore $f(x)$. Lo stesso non vale nello stile di programmazione imperativo visto fino a ora, dove i metodi e le funzioni possono produrre risultati diversi in momenti diversi, anche fornendo lo stesso input. Questo perché l'output prodotto da tali metodi e funzioni può dipendere dallo stato del

Paradigma di programmazione

Un paradigma di programmazione è l'insieme di strumenti concettuali forniti da un linguaggio di programmazione per la stesura di un programma. Un paradigma di programmazione può quindi essere visto come lo stile fondamentale con cui si concepisce e percepisce un programma.

Lambda calcolo

Le origini del paradigma di programmazione funzionale sono da ritrovarsi nel *lambda calculus* (o λ-calcolo), un sistema formale definito dal matematico Alonzo Church negli anni Trenta. Il λ-calcolo permette di modellare una computazione attraverso la composizione di funzioni, per poi analizzarne formalmente le proprietà.

programma in cui sono eseguiti (per esempio, quando tale output dipende da una variabile globale o statica, o da un attributo di un oggetto, il cui valore può variare durante l'esecuzione di un programma). Il codice funzionale è indipendente dallo stato del programma in cui è eseguito, e questo rende tale codice più facile da comprendere e da analizzare.

Un'altra importante differenza rispetto alla classica programmazione orientata agli oggetti è il supporto alle funzioni come «cittadini di prima classe». Le funzioni possono apparire ovunque all'interno di un programma, tanto quanto altri «cittadini di prima classe» come i valori (numeri, stringhe, oggetti, ecc.) e le variabili. Una funzione può quindi essere assegnata a una variabile, passata come parametro di input, o restituita come output. Questo fornisce una maggiore flessibilità al programmatore consentendogli, per esempio, di definire funzioni di ordine superiore (ovvero funzioni che prendono altre funzioni come parametri di input e/o restituiscono altre funzioni come risultato in output).

Java è un linguaggio orientato agli oggetti (e non un linguaggio funzionale), ma dalla sua versione 8 permette di strutturare parte del codice di un programma secondo il paradigma funzionale, supportando anche le funzioni di ordine superiore. L'obiettivo di questo capitolo è quello di mostrare come utilizzare il supporto funzionale messo a disposizione da Java, sfruttandolo per ottenere programmi meno verbosi e più semplici da comprendere e analizzare.

1 Funzioni: oggetti Java

Java permette di trattare le funzioni come «cittadini di prima classe», consentendo di dichiararle e utilizzarle come normali oggetti (assegnabili a variabili, fornibili come input ad altre funzioni, ecc.). Il meccanismo alla base di tutto questo è quello delle espressioni *lambda*.

Le espressioni ***lambda*** sono funzioni anonime, ovvero funzioni che non sono legate ad alcun identificatore. In Java, un'espressione *lambda* è sostanzialmente un blocco di codice preceduto dalla specifica delle variabili che devono essere passate al blocco stesso come parametri.

ESEMPIO

Si consideri la funzione che, dati due interi, ne restituisce la somma dei quadrati. La seguente espressione *lambda* rappresenta tale funzione in Java:

```
(int a, int b) -> {
  return a*a + b*b;
};
```

Un modo più compatto di esprimere la stessa funzione, sempre con una funzione *lambda*, è il seguente:

```
(a,b) -> {a*a + b*b};
```

L'esempio precedente mostra come scrivere una stessa espressione *lambda* in modo più compatto, applicando due delle seguenti regole di semplificazione:

- il tipo degli argomenti passati in input a un'espressione *lambda* può essere omesso, in quanto il compilatore può determinarlo automaticamente in base al loro utilizzo;
- se l'argomento in input è uno solo, allora i simboli «(» e «)» possono essere omessi;
- se il blocco di codice di un'espressione *lambda* consiste solamente in un'istruzione `return`, allora il blocco di codice può essere semplificato omettendo i simboli «{» e «}» e l'istruzione `return`, lasciando solamente il risultato calcolato.

Può accadere che un'espressione *lambda* consista nella sola invocazione di un metodo esistente. In tal caso, si può riferire direttamente il metodo invocato, utilizzando il meccanismo di *method reference* offerto da Java. Si possono riferire:

- i metodi statici resi disponibili da una classe (scrivendo l'espressione *nomeClasse::nomeMetodo*);
- i metodi d'istanza messi a disposizione in un oggetto (scrivendo l'espressione *nomeOggetto::nomeMetodo*);
- i metodi d'istanza resi disponibili da tutti gli oggetti di un determinato tipo (scrivendo l'espressione *nomeTipo::nomeMetodo*).

Un *method reference*, quindi, non è altro che un modo più semplice e compatto di scrivere un'espressione *lambda* che utilizzi il solo metodo riferito.

ESEMPIO

Si consideri l'espressione *lambda* che, data una stringa *s* in input, utilizza il metodo *toUpperCase* della classe *String* per restituire una copia di *s* in cui tutti i caratteri sono maiuscoli. Tale espressione si può scrivere come segue:

```
s -> s.toUpperCase()
```

Grazie al meccanismo di *method reference*, l'espressione *lambda* mostrata sopra può essere scritta anche come segue:

```
String::toUpperCase
```

Le espressioni *lambda* possono essere utilizzate come argomento da passare a una funzione (di ordine superiore), o durante la definizione di una nuova funzione. Il *package java.util.function* mette a disposizione un insieme di interfacce predefinite per la definizione e l'utilizzo di funzioni. Un sottoinsieme di tali interfacce è riportato nella TABELLA 1.

TABELLA 1

Interfaccia	Utilizzo
Function<T, R>	Dichiarazione di funzioni che accettano un argomento in input e producono un output. L'argomento in input è un oggetto di tipo *T*, mentre l'output è un oggetto di tipo *R*.
UnaryOperator<T>	Dichiarazione di funzioni che accettano un argomento in input e producono un output. L'argomento in input e l'output sono entrambi oggetti di tipo *T*. ▶

Interfaccia	Utilizzo
BiFunction<T, U, R>	Dichiarazione di funzioni che accettano due argomenti in input e producono un output. Il primo argomento in input è un oggetto di tipo *T*, mentre il secondo argomento in input è un oggetto di tipo *U*. L'output è un oggetto di tipo *R*.
BinaryOperator<T>	Dichiarazione di funzioni che accettano due argomenti in input e producono un output. Gli argomenti in input e l'output sono tutti oggetti di tipo *T*.
Predicate<T>	Dichiarazione di predicati booleani unari, ovvero funzioni che accettano un argomento in input e producono un output booleano. L'argomento in input è un oggetto di tipo *T*.
BiPredicate<T, U>	Dichiarazione di predicati booleani binari, ovvero funzioni che accettano due argomenti in input e producono un output di tipo **boolean**. Gli argomenti in input sono un oggetto di tipo *T* e un oggetto di tipo *U*.
Consumer<T>	Dichiarazione di funzioni che accettano un argomento in input e non producono alcun output. L'argomento in input è un oggetto di tipo *T*.
Supplier<T>	Dichiarazione di funzioni che non accettano argomenti in input, ma producono in output un oggetto di tipo *T*.

L'interfaccia *Function<T, R>* consente di definire funzioni che prendono in input un singolo argomento (di tipo *T*) e restituiscono un singolo output (di tipo *R*). L'insieme dei metodi disponibili in *Function<T, R>* è riportato nella TABELLA **2**.

TABELLA **2**

Firma del metodo	Descrizione delle funzionalità
`R apply(T t)`	Applica la funzione all'argomento *t* in input, e restituisce l'output calcolato (un oggetto di tipo *R*).
`Function<T, V> andThen(` `  Function<? super R,? extends V> f)`	Restituisce una funzione composta che prima applica la funzione corrente all'input, e poi applica la funzione *f* al risultato calcolato dalla funzione corrente.
`Function<V, R> compose(` `  Function<? super V,? extends T> f)`	Restituisce una funzione composta che prima applica la funzione *f* all'argomento in input, e poi applica la funzione corrente al risultato calcolato dalla funzione *f*.

ESEMPIO

Si consideri la funzione che, data una stringa *s* ricevuta in input, ne restituisce il carattere iniziale. Tale funzione può essere definita come segue:

```
Function<String, Character> iniziale = (String s) -> {
  return s.charAt(0);
};
```

Applicando le regole di semplificazione delle espressioni *lambda* è possibile ottenere una definizione più compatta della funzione *iniziale*:

```
Function<String, Character> iniziale = s -> s.charAt(0);
```

In entrambi i casi, la funzione iniziale può essere utilizzata per calcolare la lettera iniziale di una stringa semplicemente invocando il metodo *apply* (reso disponibile dalla stessa funzione iniziale):

```
System.out.println(iniziale.apply("PIPPO")); // visualizza "P"
```

UnaryOperator<T> consente di definire funzioni i cui input e output sono entrambi oggetti di tipo *T*. Poiché tale interfaccia è definita come estensione dell'interfaccia *Function<T, R>*, *UnaryOperator<T>* eredita i metodi riportati nella TABELLA **2**.

ESEMPIO Si consideri la funzione che, dato un numero *n* ricevuto in input, ne calcola il cubo. Per definire tale funzione in Java, è sufficiente scrivere quanto segue:

```
UnaryOperator<Integer> cubo = n -> (n*n*n);
```

La funzione *cubo* può poi essere utilizzata per calcolare il cubo di un numero invocando il suo metodo *apply* come mostrato di seguito:

```
System.out.println(cubo.apply(-3)); // visualizza "-27"
```

Tramite l'interfaccia *BiFunction<T, U, R>* si possono definire funzioni che prendono in input due argomenti (uno di tipo *T* e uno di tipo *U*) e restituiscono un singolo oggetto in output (di tipo *R*). L'insieme dei metodi disponibili in *BiFunction<T, U, R>* è riportato nella TABELLA 3.

TABELLA 3

Firma del metodo	Descrizione delle funzionalità
`R apply (T t, U u)`	Applica la funzione agli argomenti *t* e *u* ricevuti in input, e restituisce l'output calcolato.
`BiFunction<T, U, V> andThen( Function<? super R,? extends V> f)`	Restituisce una funzione composta che prima applica la funzione corrente all'input, e poi applica la funzione *f* al risultato calcolato dalla funzione corrente.

ESEMPIO Si consideri la funzione che, dati in input una stringa *s* e un carattere *c*, conta il numero di occorrenze di tale carattere nella stringa. Tale funzione può essere definita in Java come segue:

```
BiFunction<String, Character, Integer> occorrenze = (s, c) -> {
  int contatore = 0;
  for (int i=0; i<s.length(); i++)
    if (s.charAt(i)==c) contatore++;
  return contatore;
};
```

La funzione *occorrenze* può poi essere utilizzata per determinare il numero di occorrenze di un carattere in una stringa come segue:

```
System.out.println(occorrenze.apply("PIPPO",'P')); // visualizza "3"
```

L'interfaccia *BinaryOperator<T>* permette la definizione di funzioni aventi due argomenti in input e un oggetto in output, fissando il tipo di tali input e output a *T*. *BinaryOperator<T>* è definita estendendo *BiFunction<T, U, R>* e perciò eredita i metodi riportati nella TABELLA 3.

ESEMPIO Si consideri la funzione che permette di calcolare la velocità media di un veicolo, dati in input i chilometri percorsi e il tempo (in ore) impiegato a percorrerli. Tale funzione può essere definita come un *BinaryOperator<Double>*:

```
BinaryOperator<Double> velocità = (km, ore) -> (km/ore);
```

Supponiamo ora di voler calcolare la velocità media di un veicolo che abbia percorso 316 chilometri e 800 metri in 5 ore e mezza; tale calcolo può essere svolto utilizzando la funzione *velocità* come segue:

```
System.out.println(velocità.apply(316.8,5.5)); // visualizza "57.6"
```

L'interfaccia *Predicate<T>* consente di definire predicati booleani unari, ovvero funzioni che ricevono un argomento in input di tipo *T* e restituiscono in output un valore di tipo `boolean`. I metodi messi a disposizione da *Predicate<T>* sono elencati nella TABELLA 4.

L'interfaccia *Consumer<T>* consente di implementare funzioni che ricevono un argomento in input (di tipo *T*) e non restituiscono alcun output. Un oggetto di tipo *Consumer<T>* mette a disposizione il metodo *accept*, che permette di eseguire la funzione implementata su un dato input.

ESEMPIO

Si consideri la funzione che, data una stringa *s* ricevuta in input, la visualizza sullo schermo. Tale funzione può essere implementata come segue:

```
Consumer<String> visualizza = s -> {
   System.out.println(s);
};
```

La funzione *visualizza* può poi essere utilizzata semplicemente invocando il suo metodo *accept* come mostrato di seguito:

```
visualizza.accept("PIPPO"); // visualizza "PIPPO"
```

L'interfaccia *Supplier<T>* consente di implementare funzioni che non ricevono alcun argomento in input, ma restituiscono un oggetto in output (di tipo *T*). Tali funzioni vengono eseguite ogniqualvolta sia invocato il loro metodo *get*.

ESEMPIO

Si prenda la funzione *casuale*, la quale genera un numero casuale compreso tra 0 e 99.

```
Supplier<Integer> casuale = () -> ((int)(Math.random()*100));
```

Per generare un numero casuale è poi sufficiente invocare il metodo *get* della funzione *casuale*:

```
System.out.println(casuale.get()); // visualizza un numero casuale
```

2 Composizione di funzioni

Proprio come accade per le funzioni matematiche, le funzioni implementate in Java possono essere composte. I meccanismi di composizione supportati sono la **sequenza** e l'**annidamento**.

- Due o più funzioni possono essere composte in sequenza semplicemente utilizzando l'output di una funzione come input della successiva.
- Il meccanismo di annidamento prevede invece l'utilizzo di una o più funzioni all'interno di un'altra.

Il meccanismo di composizione in sequenza si implementa sfruttando il metodo *andThen* messo a disposizione dalle interfacce *Function<T, R>* e *BiFunction<T, U, R>* (e dalle interfacce da esse derivate, come *UnaryOperator<T>* e *BinaryOperator<T>*). Invocando il metodo *andThen* su di una funzione *f* esistente, e passando a tale metodo un'altra funzione *g* come argomento, *andThen* restituisce una nuova funzione che prima applica *f* all'input ricevuto e poi applica *g* al risultato calcolato da *f*.

ESEMPIO Si considerino le seguenti funzioni:

```
UnaryOperator<Integer> cubo = n -> (n*n*n);
UnaryOperator<Integer> doppio = n -> (2*n);
```

Dato un numero intero *n*, *cubo* calcola n^3 mentre *doppio* calcola $2n$. Tali funzioni possono essere composte in sequenza per calcolare $2n^3$ (ovvero il doppio del cubo) e $(2n)^3$ (ovvero il cubo del doppio).

```
Function<Integer, Integer> doppioDelCubo = cubo.andThen(doppio);
Function<Integer, Integer> cuboDelDoppio = doppio.andThen(cubo);
```

Le funzioni *doppioDelCubo* e *cuboDelDoppio* possono poi essere utilizzate come mostrato di seguito:

```
System.out.println(doppioDelCubo.apply(2)); // visualizza "16"
System.out.println(cuboDelDoppio.apply(2)); // visualizza "64"
```

OSSERVAZIONE L'ordine con cui due o più funzioni sono messe in sequenza determina il valore calcolato dalla funzione composta che si ottiene. Ordini diversi possono portare infatti alla definizione di funzioni diverse (come mostrato nell'esempio precedente).

Il meccanismo di annidamento permette di definire e utilizzare funzioni all'interno della definizione di una nuova funzione. In altre parole, all'interno dell'espressione *lambda* che definisce una nuova funzione, possono essere definite e/o utilizzate altre funzioni.

ESEMPIO Si consideri la funzione che calcola la somma dei valori assoluti di due numeri. La funzione *sommaValAssoluti* implementa la funzione desiderata, sfruttando il meccanismo di annidamento (ovvero definendo e utilizzando la funzione *valoreAssoluto* al suo interno).

```
BiFunction<Integer, Integer, Integer> sommaValAssoluti = (a, b) -> {
  UnaryOperator<Integer> valoreAssoluto = (n) -> {
    if(n < 0) return -n;
    return n;
  };

  return valoreAssoluto.apply(a) + valoreAssoluto.apply(b);
};
```

La funzione *sommaValAssoluti* può poi essere utilizzata come mostrato di seguito:

```
System.out.println(sommaValAssoluti.apply(4, 5));  // visualizza "9"
System.out.println(sommaValAssoluti.apply(-4, 5)); // visualizza "9"
```

OSSERVAZIONE Sebbene il meccanismo di annidamento non comporti alcun problema quando si vanno a trattare predicati booleani (ovvero quelle funzioni implementate mediante le interfacce *Predicate<T>* o *BiPredicate<T, U>*), lo stesso non vale per la composizione in sequenza. L'output di un predicato non può essere utilizzato come input di un altro predicato perché *Predicate<T>* e *BiPredicate<T, U>* non offrono il metodo *andThen*.

La composizione di predicati booleani avviene mediante AND e OR logici. Le interfacce *Predicate<T>* e *BiPredicate<T, U>* offrono infatti i metodi *and* e *or* per la generazione di predicati composti. Invocando il metodo

and su di un predicato esistente, e passando a tale metodo un altro predicato esistente, *and* restituisce un predicato composto che, dapprima applica entrambi i predicati all'input ricevuto, e poi restituisce l'AND logico dei valori booleani calcolati. Il comportamento del metodo *or* è analogo.

ESEMPIO Si considerino i predicati *pari* e *positivo*, che verificano rispettivamente se un numero *n* sia divisibile per due o maggiore di zero.

```
Predicate<Integer> pari = n -> (n % 2 == 0);
Predicate<Integer> positivo = n -> (n > 0);
```

Tali predicati possono essere composti in AND e OR come segue:

```
Predicate<Integer> pariEpositivo = pari.and(positivo);
Predicate<Integer> pariOpositivo = pari.or(positivo);
```

Il predicato composto *pariEpositivo* consente di verificare se un numero è sia pari sia positivo. Il predicato *pariOpositivo* permette, invece, di verificare se un numero soddisfi almeno una delle due condizioni. Alcuni esempi di applicazione di entrambi i predicati sono mostrati di seguito.

```
System.out.println(pariEpositivo.test(4));   // visualizza "true"
System.out.println(pariEpositivo.test(-2));  // visualizza "false"
System.out.println(pariOpositivo.test(-2));  // visualizza "true"
System.out.println(pariOpositivo.test(-3));  // visualizza "false"
```

3 Funzioni ricorsive

Proprio come accade per un algoritmo, una funzione si dice *ricorsiva* quando viene definita in termini di se stessa. Purtroppo, Java (nella sua versione 8) non supporta direttamente la definizione di funzioni ricorsive.

OSSERVAZIONE La definizione di funzioni ricorsive non va confusa con la definizione di metodi ricorsivi. I metodi di una classe possono essere sempre definiti ricorsivamente, in quanto il blocco di codice che definisce il corpo di un metodo può sempre contenere una o più invocazioni al metodo stesso. Lo stesso non vale per le funzioni, ovvero per gli oggetti definibili utilizzando le classi e le interfacce disponibili nel *package java.util.function*.

ESEMPIO Un classico esempio di funzione che può essere definita ricorsivamente è dato dalla funzione *fattoriale*. Dato un numero *n*, tale funzione calcola il valore *n*! (ovvero il prodotto dei primi *n* numeri naturali):

$$n! = n \cdot (n-1) \cdot (n-2) \cdot _ \cdot 1$$

La funzione *fattoriale* può essere definita ricorsivamente poiché il valore di *n*! si può calcolare come il prodotto $n \cdot (n-1)!$ (ponendo $0! = 1$). Si potrebbe perciò pensare di implementare ricorsivamente la funzione *fattoriale* come spiegato di seguito:

- se *n* è uguale a zero, la funzione restituisce il valore di 0! (ovvero 1).
- se *n* è maggiore di zero, invece, la funzione restituisce il valore del prodotto tra *n* ed $(n-1)!$ (che può essere calcolato applicando la funzione *fattoriale* a $n-1$).

```
Function<Integer, Integer> fattoriale = n -> {
  if(n == 0)
    return 1;
```

```
    else
      return n*fattoriale.apply(n-1);
  };
```

L'implementazione mostrata sopra, però, risulta non valida. Infatti, si ottiene il seguente errore:

```
variable fattoriale might not have been initialized
```

L'errore mostrato sopra può essere aggirato rinunciando a una definizione ricorsiva della funzione *fattoriale*, e definendo il corpo della funzione stessa in stile imperativo.

```
Function<Integer, Integer> fattoriale = n -> {
  int risultato = 1;
  while(n>0) {
    risultato = risultato*n;
    n--;
  }
  return risultato;
};
```

OSSERVAZIONE L'esempio precedente permette di apprezzare nuovamente come un'implementazione funzionale sia più comprensibile e leggibile di un'equivalente implementazione imperativa. Supponiamo, infatti, di non sapere quale sia la funzione implementata negli esempi precedenti:

- semplicemente leggendo il codice dell'implementazione funzionale, ci si accorge rapidamente che la funzione implementata è $n!$ (in quanto il codice ricalca precisamente la definizione matematica di $n!$);
- l'implementazione iterativa, invece, richiede di simulare una o più invocazioni per comprendere quale sia la funzione calcolata.

La seconda implementazione risulta quindi più difficile da leggere e comprendere della prima.

Una funzione viene trattata in Java come un oggetto, che può essere assegnato a una variabile. Di conseguenza, poiché Java non consente di utilizzare una variabile fino a che non le sia stato assegnato un valore o un oggetto, non è possibile utilizzare la variabile associata alla funzione che si sta definendo nel corpo di tale definizione.

La definizione di funzioni ricorsive in Java richiede quindi di aggirare questo problema. Una possibile soluzione è quella di definire e utilizzare opportunamente una funzione di supporto che chiameremo *helper*, annidata nella *funzioneRicorsiva* che si vuole definire.

```
Function<T, R> funzioneRicorsiva = t -> {
  BiFunction<BiFunction, T, R> helper = (f, x) -> {
    /*
      corpo della funzione ricorsiva (dove tutte le
      invocazioni a "funzioneRicorsiva" sono rimpiazzate
      da invocazioni a "f")
    */
  };
  return helper.apply(helper,t);
};
```

La funzione *helper* sfrutta il fatto che le funzioni sono «cittadini di prima classe».

- Tale funzione prende infatti in input una funzione *f* e un valore *x* e definisce il corpo di *funzioneRicorsiva* rimpiazzando tutte le invocazioni a *funzioneRicorsiva* con invocazioni alla funzione *f* passata come parametro.
- La definizione si completa poi invocando la funzione *helper*, a cui vengono passati la funzione *helper* stessa e il valore *t* passato in input a *funzioneRicorsiva*.

Così invocata, *helper* calcolerà il valore in output utilizzando la funzione *helper* stessa (in quanto ogni invocazione di *f* nel corpo di *helper* risulterà in un'invocazione di *helper* stessa). Poiché *funzioneRicorsiva* corrisponde all'invocazione ricorsiva di *helper*, tale funzione può essere considerata una funzione ricorsiva.

OSSERVAZIONE Una funzione così definita non presenta alcun problema di validità, per i seguenti motivi:

- la funzione *helper* viene definita utilizzando solamente i parametri *f* e *x* forniti in input all'espressione *lambda* che definisce il corpo di *helper* stessa;
- le variabili passate in input all'invocazione di *helper* sono note a tempo di invocazione (in quanto una è la funzione *helper* stessa e l'altra è il parametro passato in input a *funzioneRicorsiva*).

ESEMPIO Si consideri nuovamente la funzione *fattoriale*. Tale funzione può essere implementata come una funzione che prende in input un intero *n* e restituisce l'intero corrispondente al valore di *n*!:

```
Function<Integer, Integer> fattoriale = n -> {
  BiFunction<BiFunction, Integer, Integer> helper = (f, x) -> {
    if(x == 0)
      return 1;
    else
      return x*(int)f.apply(f, x-1);
  };
  return helper.apply(helper, n);
};
```

L'implementazione di *fattoriale* definisce e utilizza una funzione *helper*, il cui corpo definisce il calcolo di *x*!:

- se *x* è uguale a zero, restituisce il valore di 0! (ovvero 1).
- Se *x* è maggiore di zero, invece, restituisce il valore del prodotto $x \cdot (x - 1)!$.

La funzione *helper* viene poi invocata passandogli come argomenti la stessa funzione *helper* e il valore *n* di cui calcolare il fattoriale. Il risultato di tale invocazione viene restituito come risultato della funzione *fattoriale*.
La funzione *fattoriale* può essere utilizzata per calcolare (ricorsivamente) il valore di un numero. Tale calcolo richiede semplicemente di invocare il metodo *apply* offerto da *fattoriale*, e di passare a tale metodo il valore di cui calcolare il fattoriale.

```
System.out.println(fattoriale.apply(5)); // visualizza "120"
```

4 Applicazione di funzioni a collezioni di oggetti

Lo stile funzionale risulta particolarmente efficace quando applicato a collezioni di oggetti. Gli oggetti contenuti in una collezione possono essere infatti acceduti sotto forma di *stream* (o sequenza di oggetti) e processati applicandovi una o più funzioni.

Per processare una collezione di oggetti sotto forma di *stream*, è sufficiente invocare il metodo *stream* offerto dalla collezione stessa (purché tale collezione implementi l'interfaccia *Collection<T>* o un'interfaccia da essa derivata). Tale invocazione risulta nella creazione di un oggetto della classe *Stream<T>*, che consente di accedere e processare tutti gli oggetti presenti in una sorgente (nel nostro caso, una collezione di oggetti).

- Gli oggetti in uno *Stream<T>* non sono accessibili o processabili singolarmente, in quanto vengono processati tutti allo stesso modo (per esempio, applicando a ciascuno di essi la stessa funzione).
- Le operazioni effettuate sugli oggetti contenuti in uno *Stream<T>* non modificano gli oggetti originali.
- Un'operazione eseguita su uno *Stream<T>* può restituire un nuovo *Stream<T>* contenente i risultati calcolati, a sua volta processabile mediante un'altra operazione.

I principali metodi offerti da *Stream<T>* sono riportati nella TABELLA 5.

TABELLA 5

Firma del metodo	Descrizione delle funzionalità
`boolean allMatch(Predicate<? super T> p)`	Verifica se tutti gli oggetti contenuti in uno *Stream<T>* soddisfano il predicato *p*.
`boolean anyMatch(Predicate<? super T> p)`	Verifica se almeno uno degli oggetti contenuti in uno *Stream<T>* soddisfa il predicato *p*.
`boolean noneMatch(Predicate<? super T> p)`	Verifica che nessuno degli oggetti contenuti in uno *Stream<T>* soddisfi il predicato *p*.
`void forEach(Consumer<? super T> azione)`	Esegue l'*azione* indicata su tutti gli oggetti nello *stream*.
`Stream<T> filter(Predicate<? super T> p)`	Restituisce un nuovo *Stream<T>* contenente tutti gli oggetti nello *Stream<T>* corrente che soddisfano il predicato *p*.
`Stream<R> map(Function<? super T, ? extends R> f)`	Restituisce un nuovo *Stream<T>* contenente i risultati ottenuti applicando la funzione *f* a ciascun oggetto nello *Stream<T>* corrente.
`T reduce(T neutro, BinaryOperator<T> aggregatore)`	Riduce gli oggetti contenuti in uno *Stream<T>* a un unico oggetto utilizzando la funzione (associativa) *aggregatore* e l'elemento *neutro* di tale funzione.
`long count()`	Restituisce il numero di elementi contenuti nello *Stream<T>* cui è applicato.

I metodi *allMatch*, *anyMatch* e *noneMatch* consentono di verificare se tutti, alcuni, o nessuno degli oggetti di una collezione soddisfano una determinata condizione. Tale condizione deve essere espressa definendo un corrispondente predicato booleano (ovvero definendo un *Predicate<T>*).

ESEMPIO Si consideri l'implementazione di un metodo *contienePari* che, ricevuta in input una lista di numeri interi, verifica se almeno uno degli elementi di tale lista è *pari*. Il metodo *contienePari* può essere implementato come segue:

```
public static boolean contienePari(List<Integer> numeri) {
  Predicate<Integer> pari = n -> ((n % 2) == 0);
  return numeri.stream().anyMatch(pari);
}
```

Lo stesso metodo, senza il supporto funzionale offerto da *Predicate<T>* e *Stream<T>*, avrebbe richiesto un'implementazione molto meno concisa.

```
public static boolean contienePari(List<Integer> numeri) {
  for (int n: numeri) {
    if (n % 2 == 0) return true;
  }
  return false;
}
```

OSSERVAZIONE L'esempio precedente evidenzia chiaramente la differenza fra lo stile dichiarativo della programmazione funzionale e quello imperativo della programmazione orientata agli oggetti classica.

- L'implementazione funzionale indica solamente cosa debba essere verificato, ovvero che almeno uno degli oggetti della collezione soddisfi il predicato definito.
- La seconda implementazione invece richiede di descrivere come effettuare tale verifica, ovvero come iterare sugli oggetti della collezione e cosa fare con ciascuno di tali oggetti.

Il metodo *forEach* consente di eseguire una determinata azione su ciascuno degli elementi di uno *Stream<T>*. Tale azione deve essere implementata come un oggetto di tipo *Consumer<T>*, ovvero come una funzione che accetta un argomento in input e non restituisce alcun argomento in output.

ESEMPIO Si consideri la funzione che, data una lista di stringhe, le visualizza tutte sullo schermo. Di seguito sono elencate due sue implementazioni (funzionali) equivalenti. La prima implementazione accede alla lista di stringhe sotto forma di *Stream<T>*.

```
public static void visualizza(List<String> stringhe) {
  stringhe.stream().forEach(System.out::println);
}
```

La seconda implementazione, invece, utilizza il metodo *forEach* messo a disposizione direttamente dall'interfaccia *List<T>*.

```
public static void visualizza(List<String> stringhe) {
  stringhe.forEach(System.out::println);
}
```

Anche questo esempio permette di evidenziare lo stile dichiarativo della programmazione funzionale e di confrontarlo con quello imperativo della programmazione orientata agli oggetti:

- entrambe le implementazioni, infatti, permettono di definire cosa deve essere fatto (ovvero applicare la funzione di visualizzazione a ciascuna delle stringhe); ▶

- un'implementazione in stile imperativo, invece, richiede di descrivere il procedimento necessario a visualizzare il contenuto della lista *stringhe*. In quest'ottica, una possibile soluzione è quella di iterare su ciascuna delle stringhe in essa contenute, e applicare a ciascuna iterazione la funzione di visualizzazione.

```
public static void visualizza(List<String> stringhe) {
  for (String s: stringhe) {
    System.out.println(s);
  }
}
```

OSSERVAZIONE Il metodo *forEach* può essere applicato anche direttamente a una collezione, in quanto viene esposto dall'interfaccia *Collection<T>* (e da tutte le interfacce da essa derivate). Ciò significa che, quando la sola operazione da svolgere su di una collezione è *forEach*, non è necessario accedere agli oggetti contenuti in tale collezione sotto forma di *Stream<T>*.

Il metodo *filter* consente di filtrare gli oggetti contenuti in uno *Stream<T>*, preservando solo gli oggetti che soddisfano una determinata condizione (definita opportunamente come predicato booleano). Tali oggetti vanno a comporre un nuovo *Stream<T>*, che viene restituito da *filter* e può essere processato a sua volta.

Fluent API interface

Java supporta le *fluent interface*, ovvero la possibilità di costruire API che consentono di invocare una serie di metodi in cascata. Ogni invocazione risulta nella restituzione di un oggetto, che va a costituire il contesto dell'invocazione successiva. Per esempio, scrivendo *o.m1().m2()*, si invoca dapprima il metodo *m1* reso disponibile dall'oggetto *o*, e poi si invoca il metodo *m2* reso disponibile dall'oggetto restituito dall'invocazione di *m1*.

ESEMPIO

Si consideri la funzione che, data una lista di numeri interi, restituisce una lista che contiene i soli numeri positivi. Tale funzione può essere implementata utilizzando *filter* e *forEach* come segue:

```
public static List<Integer> soloPositivi(List<Integer> numeri) {
  Predicate<Integer> positivo = n -> (n > 0);
  List<Integer> positivi = new ArrayList<Integer>();
  numeri.stream()
        .filter(positivo)
        .forEach(n -> positivi.add(n));
  return positivi;
}
```

L'espressione *lambda* passata come argomento al *forEach* è equivalente al metodo *add* offerto dall'interfaccia *List<T>*. Possiamo evitare di ridefinirlo come espressione *lambda*, in quanto Java consente di fornire come argomento di *forEach* direttamente un riferimento al metodo *add* dell'oggetto *positivi* (utilizzando il meccanismo di *method reference*, ovvero la sintassi *positivi::add*).

```
public static List<Integer> soloPositivi(List<Integer> numeri) {
  Predicate<Integer> positivo = n -> (n > 0);
  List<Integer> positivi = new ArrayList<Integer>();
  numeri.stream()
        .filter(positivo)
        .forEach(positivi::add);
  return positivi;
}
```

Entrambe le implementazioni, in ogni caso, processano la lista di numeri applicandovi una sequenza di operazioni. Dopo aver creato lo *Stream<T>* (invocando il metodo *stream*), il metodo *filter* costruisce un nuovo *Stream<T>* contenente tutti e solo i numeri positivi contenuti nello *Stream<T>* iniziale. Successivamente, il metodo *forEach* è utilizzato per costruire la lista *positivi* restituita dal metodo implementato.

Uno *Stream<T>* viene generalmente processato applicandovi una sequenza di operazioni, l'output di ciascuna delle quali è utilizzato come input della successiva. La sequenza di operazioni viene comunemente denominata *pipeline*, e ciascuna operazione costituisce uno stadio di tale *pipeline*.

- Una *pipeline* ha sempre un solo stadio iniziale (s_0), che si occupa di generare lo *Stream<T>* di oggetti da processare. Lo stadio iniziale è l'unico stadio della *pipeline* dotato del solo canale di output.
- Vi sono poi una serie di stadi intermedi ($s_1 ... s_{n-1}$), dotati sia di un canale di input (collegato all'output dello stadio precedente) sia di un canale di output (collegato all'input dello stadio successivo). Ciascuno di tali stadi riceve uno *Stream<T>* sul canale di input, applica la funzione per cui è progettato a ciascuno degli oggetti di tale *Stream<T>* e restituisce (sul canale di output) un nuovo *Stream<T>* contenente i risultati calcolati.
- Una *pipeline* ha sempre un solo stadio finale (s_n), che si occupa di consumare lo *Stream<T>* contenente i risultati calcolati dall'ultimo degli stadi intermedi. Lo stadio finale è l'unico stadio della *pipeline* dotato del solo canale di output.

Per quanto visto fino a ora, il metodo *stream* offerto dalle collezioni può svolgere il compito di stadio iniziale di una *pipeline*, in quanto genera uno *Stream<T>* a partire da una collezione di oggetti. Il metodo *filter*, invece, è un chiaro esempio di stadio intermedio di una *pipeline*, visto che riceve in input uno *Stream<T>* e restituisce in output un nuovo *Stream<T>*. Il metodo *forEach* può essere utilizzato come stadio finale di una *pipeline*, in quanto processa tutti gli elementi dello *Stream<T>* che riceve in input applicandovi una funzione di tipo *Consumer<T>* (e quindi non restituendo oggetti che vadano a formare un nuovo *Stream<T>* in output). Anche i metodi *allMatch*, *anyMatch* e *noneMatch* possono essere utilizzati come stadi finali di una *pipeline*, dato che ciascuno di essi processa lo *Stream<T>* ricevuto in input verificando una determinata condizione e restituendo solamente un valore booleano come risultato complessivo.

Il metodo *map* consente di applicare una funzione a ciascun oggetto contenuto in uno *Stream<T>*, e di costruire un nuovo *Stream<T>* contenente i risultati. Può quindi essere utilizzato come stadio intermedio di una *pipeline*, per trasformare gli oggetti ricevuti in input mediante la funzione che vi viene applicata.

ESEMPIO Si prenda in considerazione il seguente metodo *main*, il quale prima costruisce una lista di stringhe, e poi le visualizza sullo schermo (separate da spazi). Tale metodo è implementato utilizzando una *pipeline* costituita dai soli stadi iniziale (*stream*) e finale (*forEach*).

```
public static void main(String[] args) {
  List<String> stringhe = Arrays.asList("Banana", "Mela", "Pera");
  stringhe.stream()
          .forEach(s -> System.out.print(s + " "));
}
```

L'output visualizzato sullo schermo è il seguente:

```
Banana Mela Pera
```

Inserendo uno stadio intermedio nella *pipeline* soprastante, è possibile modificare le parole visualizzate sullo schermo, rendendole tutte maiuscole. Ciò si ottiene facilmente invocando il metodo *map* come mostrato di seguito.

```
public static void main(String[] args) {
  List<String> stringhe = Arrays.asList("Banana", "Mela", "Pera");
  stringhe.stream()
          .map(String::toUpperCase)
          .forEach(s -> System.out.print(s + " "));
}
```

In questo modo, l'output visualizzato sullo schermo diventa:

```
BANANA MELA PERA
```

Il metodo *reduce* può essere utilizzato come stadio finale di una *pipeline*. Tale metodo riduce tutti gli oggetti contenuti nello *Stream<T>* cui è applicato a un unico oggetto. Per farlo utilizza una funzione di aggregazione (*aggregatore*, nella firma mostrata nella TABELLA 5). Il compito di tale funzione è quello di combinare tutti gli oggetti contenuti in uno *Stream<T>* e restituire un unico oggetto che rappresenti il risultato di tale combinazione. Il metodo *reduce* deve essere fornito anche dell'elemento *neutro* della funzione *aggregatore*, ovvero di quel valore tale che *aggregatore(neutro, o)=o*, per un qualunque oggetto *o*.

ESEMPIO

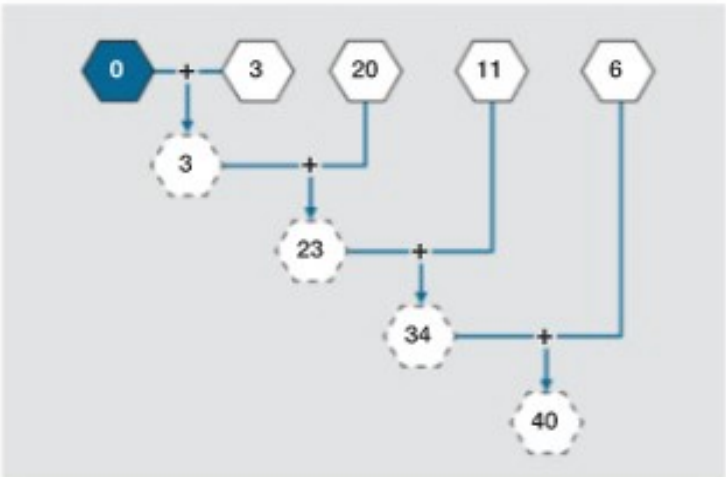
Si consideri il problema della sommatoria di una lista di numeri. Tale problema può essere risolto utilizzando il metodo *reduce*, a cui vengono passati l'espressione *lambda* che calcola la somma di due numeri e il valore zero (in quanto tale valore costituisce l'elemento neutro della somma).

```
public static int sommatoria(List<Integer> numeri) {
  return numeri.stream().reduce(0, (a, b) -> a+b);
}
```

Per capire meglio come funzioni il metodo *reduce*, è necessario analizzarne un'esecuzione effettiva (come quella del seguente metodo *main*).

```
public static void main(String[] args) {
  List<Integer> numeri = Arrays.asList(3,20,11,6);
  System.out.println("La somma è " + sommatoria(numeri));
}
```

L'esecuzione di *reduce* nel metodo *sommatoria* procede come mostrato di seguito:



Il processo di riduzione comincia sommando il primo numero della lista con zero, l'elemento neutro della somma. Il secondo numero viene poi sommato al risultato ottenuto, e il risultato di tale somma viene a sua volta sommato con il terzo numero. Si procede così fino a che non si arriva all'ultimo numero della lista. Il risultato ottenuto sommando quanto accumulato con l'ultimo numero della lista viene poi restituito dal metodo *reduce*.
L'output visualizzato sullo schermo dal metodo *main* mostrato sopra è il seguente:

```
La somma è 40
```

OSSERVAZIONE Come mostrato dall'esempio precedente, l'elemento neutro è fondamentale per far iniziare il processo di combinazione portato avanti dal metodo *reduce*. Tale elemento è infatti combinato con il primo elemento dello *Stream<T>* cui *reduce* è applicato per generare un primo risultato e poi combinarlo con i successivi elementi contenuti nello *Stream<T>*.

Quando si processa una collezione di oggetti, i metodi *filter*, *map* e *reduce* sono spesso utilizzati insieme.

- I metodi *filter* e *map* sono utilizzati come stadi intermedi di una *pipeline*. In particolare, il metodo *filter* consente di restringere gli oggetti da processare ai soli interessanti, mentre il metodo *map* consente di manipolare tali oggetti.
- Il metodo *reduce* viene utilizzato come stadio finale della *pipeline* per ottenere il risultato complessivo del calcolo.

MapReduce

Con *MapReduce* si intende una tecnica di programmazione tipicamente utilizzata per elaborare (in parallelo) grandi insiemi di dati. Tale tecnica è basata su due fasi (*Map* e *Reduce*).

- La fase *Map* suddivide i dati in sottoinsiemi. Tali sottoinsiemi sono poi elaborati parallelamente applicando una o più funzioni e producendo risultati separati.
- La fase *Reduce* combina opportunamente i risultati prodotti dalla fase *Map* applicando una funzione di aggregazione e restituendo un unico risultato finale.

ESEMPIO Si consideri il seguente problema: data una lista di numeri si vuole calcolare la somma dei quadrati di tutti e solo i numeri pari. Tale calcolo può essere svolto con il seguente metodo:

```
public static int sommaQuadratiDeiPari(List<Integer> numeri) {
  return numeri.stream()
               .filter(n -> ((n % 2) == 0))
               .map(n -> n*n)
               .reduce(0, (n, m) -> n + m);
}
```

Dopo aver invocato il metodo *stream*, *sommaQuadratiDeiPari* utilizza *map*, *filter* e *reduce* come elencato di seguito:

1. utilizza *filter* per restringere i numeri da considerare ai soli numeri pari;
2. utilizza *map* per calcolare il quadrato di tutti i numeri pari;
3. utilizza *reduce* per sommare tutti i quadrati calcolati.

Vale inoltre la pena osservare che, seppure l'esempio sia più articolato dei precedenti, lo stile dichiarativo della programmazione funzionale consente di esprimere solamente cosa deve essere fatto. Il codice risultante è compatto e semplice da verificare e comprendere. Lo stesso non si può dire del seguente metodo, che esprime la stessa computazione in stile imperativo e senza sfruttare il supporto funzionale offerto da Java.

```
public static int sommaQuadratiDeiPari(List<String> numeri) {
  int risultato = 0;
  for (int n: numeri) {
      if (n % 2 == 0) risultato += n*n;
  }
  return risultato;
}
```

In alternativa a *reduce*, è possibile utilizzare il metodo *count* come stadio finale di una *pipeline*. Il metodo *count* consente di contare quanti oggetti sono contenuti nello *Stream<T>* cui viene applicato. Ciò risulta particolarmente utile quando, per esempio, si voglioni contare quanti oggetti sono rimasti in uno *Stream<T>* dopo che vi sono stati applicati uno o più filtri (mediante il metodo *filter*).

ESEMPIO

Si prenda in esame il seguente metodo *main*, il quale prima costruisce una lista di nomi, poi conta quanti di essi iniziano con la lettera *s*, e infine visualizza sullo schermo il risultato del conteggio.

```
public static void main(String[] args) {
  List<String> nomi =
    Arrays.asList("Stefania", "Francesco", "Jacopo", "Sara");

  Predicate<String> iniziaPerS = str -> str.startsWith("s");
  long quanti = nomi.stream()
                    .map(String::toLowerCase)
                    .filter(iniziaPerS)
                    .count();

  System.out.println(quanti + " nomi iniziano con la lettera s.");
}
```

Si osservi il modo in cui viene calcolata la quantità di nomi che iniziano con la lettera *s*. Dopo aver definito un predicato *iniziaPerS* (il cui scopo è quello di verificare se una stringa inizi effettivamente con la lettera *s*), viene calcolato il valore della variabile *quanti* come segue:

1. il metodo *stream* trasforma la lista di nomi in uno *Stream<String>*;
2. il metodo *map* normalizza i nomi contenuti in tale *Stream<String>* rendendo tutte le loro lettere minuscole;
3. il metodo *filter* (assieme al predicato *iniziaPerS*) restringe lo *Stream<String>* ottenuto al passo precedente mantenendo i soli nomi che iniziano con la lettera *s*;
4. il metodo *count* conta quanti nomi sono presenti nello *Stream<String>* restituito dal metodo *filter* applicato al passo precedente.

L'output visualizzato sullo schermo eseguendo il metodo *main* mostrato sopra è quindi il seguente:

```
2 nomi iniziano con la lettera s.
```

5 Riuso mediante *lexical scoping* e *lexical closure*

La semplicità e la compattezza delle espressioni *lambda* potrebbero indurre uno sviluppatore alla duplicazione della definizione di una o più funzioni qualora sia necessario svilupparne una analoga.

ESEMPIO

Si consideri una lista di numeri e si supponga di voler determinare quanti di questi numeri siano multipli di 3 e quanti siano multipli di 4. Una possibile soluzione è quella di definire due *Predicate<Integer>* (*multiploDi3* e *multiploDi4*) e utilizzare tali predicati per filtrare e contare quanti numeri sono *multipliDi3* e quanti sono *multipliDi4*.

```
public static void main(String[] args) {
  List<Integer> numeri = Arrays.asList(24,7,15,16,28,8,40);
```

```
    Predicate<Integer> multiploDi3 = n -> ((n % 3) == 0);
    Predicate<Integer> multiploDi4 = n -> ((n % 4) == 0);
    long multipliDi3 = numeri.stream()
                             .filter(multiploDi3)
                             .count();
    long multipliDi4 = numeri.stream()
                             .filter(multiploDi4)
                             .count();

    System.out.println(multipliDi3 + " multipli di 3.");
    System.out.println(multipliDi4 + " multipli di 4.");
}
```

Il metodo *main* mostrato sopra si comporta proprio come anticipato e una sua esecuzione porta alla visualizzazione sullo schermo del seguente output:

```
2 multipli di 3.
5 multipli di 4.
```

Si osservino con attenzione i due predicati definiti: il predicato *multiploDi4* è sostanzialmente una replica del predicato *multiploDi3*, con la sola differenza dell'utilizzo del valore 4 (invece di 3) come argomento a destra dell'operatore %.

La replicazione dell'espressione *lambda* che definisce una funzione, però, non è una buona prassi da seguire. Tale replicazione può portare infatti alla generazione di codice di scarsa qualità, specialmente in termini di manutenibilità. Si consideri, per esempio, il caso in cui si vuole modificare il corpo di un'espressione *lambda* con un'implementazione più efficiente. Se tale espressione *lambda* è stata replicata in più definizioni di funzione, ciò significa che vanno ritrovate tutte le repliche e modificate tutte allo stesso modo (con un procedimento di per sé complicato e che può indurre a errori). Lo stesso problema non si presenta se l'espressione *lambda* viene scritta solo una volta per essere poi condivisa tra più definizioni di funzione.

ESEMPIO

Per evitare di replicare l'espressione *lambda* usata per definire i predicati (unari) *multiploDi3* ed *multiploDi4* si potrebbe pensare di implementare un unico predicato binario *multiploDi*. Tale predicato potrebbe infatti prendere in input due numeri *n* ed *m* e verificare se *n* sia multiplo di *m*.

```
BiPredicate<Integer, Integer> multiploDi = (n, m) -> ((n%m)==0);
```

Quanto mostrato sopra è corretto in Java, ma non consente di risolvere il problema considerato. L'obiettivo è infatti quello di sviluppare un predicato da fornire al metodo *filter*, e un *BiPredicate<Integer, Integer>* non assolve a questa funzione (in quanto *filter* accetta solamente predicati di tipo *Predicate<T>*).
L'idea di utilizzare una soluzione parametrica sia in *n* sia in *m* è però valida. Un'alternativa a quanto visto sopra è quella di costruire un metodo statico che, dato un numero *m*, restituisca un predicato (di tipo *Predicate<Integer>*) che verifichi se un numero è multiplo di *m*.

```
public static Predicate<Integer> multiploDi(int m) {
    Predicate<Integer> multiploDiM = n -> ((n % m) == 0);
    return multiploDiM;
}
```

Tale metodo può poi essere utilizzato all'interno del *main*, visto nell'esempio precedente, per evitare di replicare la definizione dei predicati *multiploDi3* e *multiploDi4*.

▶

```
public static void main(String[] args) {
  List<Integer> numeri = Arrays.asList(24,7,15,16,28,8,40);

  long multipliDi3 = numeri.stream()
                           .filter(multiploDi(3))
                           .count();
  long multipliDi4 = numeri.stream()
                           .filter(multiploDi(4))
                           .count();

  System.out.println(multipliDi3 + " multipli di 3.");
  System.out.println(multipliDi4 + " multipli di 4.");
}
```

La definizione di *multiploDi3* e *multiploDi4* viene rimpiazzata dall'invocazione del metodo statico *multiploDi*. Quando tale metodo viene invocato, infatti, restituisce direttamente un predicato che verifica se un numero sia multiplo del valore passatogli in input. Il risultato dell'invocazione di *multiploDi* può quindi essere passato direttamente in input al metodo *filter*, in quanto questo corrisponde a fornire *filter* del predicato desiderato (ovvero prima di un predicato che verifica se un numero sia multiplo di 3 e poi di un predicato che verifica se un numero sia multiplo di 4).

OSSERVAZIONE L'esempio precedente mostra un ulteriore utilizzo delle funzioni di ordine superiore presenti in Java. Mentre gli altri esempi visti in questo capitolo facevano vedere come utilizzare le funzioni come input di un metodo o di un'altra funzione, l'esempio precedente mostra come una funzione possa anche essere restituita in output.

Il predicato ritornato dal metodo *multiploDi* mostrato nell'esempio precedente, però, è diverso da quanto visto fino a ora. Mentre risulta chiaro cosa sia la variabile *n* (in quanto tale variabile costituisce l'input dell'espressione *lambda* che definisce il predicato ritornato), non è chiaro quale sia il valore associato alla variabile *m* utilizzata a destra dell'operatore %. Poiché la variabile *m* non è presente nel contesto (*scope*) dell'espressione *lambda* in cui appare, il valore di *m* è da ricercarsi nel contesto in cui tale espressione *lambda* viene definita (ovvero nel contesto del metodo *multiploDi*, dove appare come parametro passato in input a tale metodo).

Questo tipo di ricerca del valore di una variabile, che prevede di risalire i contesti in cui appare fino a che non se ne trova la definizione, viene detto ***lexical scoping***. Il *lexical scoping* è il meccanismo che consente di creare una o più variabili in un contesto (per esempio, in una classe o in un metodo) per poi utilizzarle in contesti annidati all'interno di quello di definizione (per esempio, nella definizione di una funzione o di un'espressione *lambda* presenti all'interno della classe o del metodo che definiscono le variabili utilizzate). Inoltre, quando un'espressione *lambda* utilizza variabili non locali, ovvero variabili prese dal contesto in cui è definita (come la variabile *m* nell'esempio precedente), si dice che tale espressione *lambda* esegue una ***lexical closure*** (o *chiusura lessicale*) delle variabili non locali.

OSSERVAZIONE La chiusura lessicale è un concetto presente in altri linguaggi di programmazione come, per esempio, JavaScript.

Un'espressione *lambda* può chiudere lessicalmente variabili non locali solo se sono dichiarate come `final` o se si comportano come tali. In altre parole, le regole da rispettare per poter chiudere lessicalmente una variabile non locale in un'espressione *lambda* sono le seguenti:

- una variabile non locale può essere chiusa solo se è già stata inizializzata;
- il valore associato a una variabile chiusa all'interno di un'espressione *lambda* non deve variare in seguito alla sua chiusura.

Ogni tentativo di chiudere una variabile non locale che violi una delle due condizioni indicate sopra genererà un errore di compilazione.

L'utilizzo di funzioni di ordine superiore e di *lexical scoping* consente quindi di evitare la duplicazione dell'espressione *lambda* utilizzata per definire due o più funzioni.

La soluzione mostrata sopra richiede di definire un metodo statico ogni volta che è necessario rendere parametrica una funzione. Di conseguenza, se la risoluzione di un problema richiede la definizione di più funzioni parametriche, l'implementazione Java corrispondente richiederà la definizione di più metodi statici.

Per quanto l'approccio presentato sopra sia valido, è opportuno limitare la definizione delle funzioni alle sole classi o ai soli metodi che ne fanno uso. In questa prospettiva, una valida alternativa è la definizione di funzioni il cui output sia a sua volta una funzione.

ESEMPIO Un'implementazione equivalente del metodo *main* mostrato nell'esempio precedente (che non richieda la definizione del metodo statico *multiploDi*) è la seguente.

```
public static void main(String[] args) {
  List<Integer> numeri = Arrays.asList(24,7,15,16,28,8,40);

  Function<Integer, Predicate<Integer>> multiploDi = m -> {
    Predicate<Integer> multiploDiM = n -> ((n % m) == 0);
    return multiploDiM;
  };
  long multipliDi3 = numeri.stream()
                           .filter(multiploDi.apply(3))
                           .count();
  long multipliDi4 = numeri.stream()
                           .filter(multiploDi.apply(4))
                           .count();

  System.out.println(multipliDi3 + " multipli di 3.");
  System.out.println(multipliDi4 + " multipli di 4.");
}
```

Tale implementazione prevede la definizione di una funzione *multiploDi* che, dato in input un numero *m*, restituisce il predicato che verifica se un numero è multiplo di *m*. Il corpo dell'espressione *lambda* che definisce *multiploDi* è del tutto analogo a quello dell'omonimo metodo statico presentato nell'esempio precedente. La funzione *multiploDi* può essere poi utilizzata per passare il predicato necessario alle invocazioni del metodo *filter*, utilizzando opportunamente il metodo *apply* messo a disposizione dalla stessa funzione *multiploDi*.
Se si osserva l'espressione *lambda* che definisce la funzione *multiploDi*, si può notare che tale espressione sostanzialmente consiste di due espressioni *lambda* annidate l'una dentro l'altra:

- la prima espressione *lambda* prende in input *m* e definisce e restituisce il predicato *multiploDiM*;
- la seconda espressione *lambda* è quella che definisce il predicato *multiploDiM*.

L'associazione del nome *multiploDiM* al predicato restituito è però del tutto superflua. È infatti sufficiente restituire direttamente l'espressione *lambda* che definisce tale predicato semplificando la definizione della funzione *multiploDi* come segue:

```
Function<Integer, Predicate<Integer>> multiploDi = m -> {
  return n -> ((n % m) == 0);
};
```

La funzione mostrata sopra è del tutto equivalente a quella mostrata nel metodo *main* presentato all'inizio di questo esempio. Tale funzione può essere ulteriormente semplificata, in quanto la sintassi delle espressioni *lambda* non richiede l'utilizzo dei simboli «{» e «}» se il solo costrutto utilizzato è **return**.

```
Function<Integer, Predicate<Integer>> multiploDi =
        m -> (n -> ((n%m)==0));
```

Ciò significa che il metodo *main* mostrato all'inizio di questo esempio può essere riscritto in forma più compatta come mostrato di seguito.

```
public static void main(String[] args) {
  List<Integer> numeri = Arrays.asList(24,7,15,16,28,8,40);

  Function<Integer, Predicate<Integer>> multiploDi =
          m -> n -> ((n%m)==0);
  long multipliDi3 = numeri.stream()
                           .filter(multiploDi.apply(3))
                           .count();
  long multipliDi4 = numeri.stream()
                           .filter(multiploDi.apply(4))
                           .count();

  System.out.println(multipliDi3 + " multipli di 3.");
  System.out.println(multipliDi4 + " multipli di 4.");
}
```

Infine, vale la pena osservare che non vi sono limiti al numero di simboli «->» che possono essere utilizzati durante la definizione di un'espressione. Si può infatti definire un'espressione *lambda* che restituisce un'altra espressione *lambda*, che a sua volta restituisce un'espressione *lambda*, e così via. L'unico vincolo da rispettare è quello imposto dalle regole di *lexical scoping* elencate sopra.

ESEMPIO

Si consideri una funzione *f* che, dati tre numeri interi (*a*, *b* e *c*), calcola la somma dei loro quadrati (ovvero $a^2 + b^2 + c^2$). Tale funzione può essere implementata come segue:

```
Function<Integer, Function<Integer, Function<Integer, Integer>>> f =
        a -> b -> c -> a*a + b*b + c*c;
```

La funzione *f* è definita mediante tre espressioni *lambda*. L'espressione più esterna è quella che prende in input la variabile *a* e restituisce un'altra espressione *lambda*. Tale espressione prende in input la variabile *b* e restituisce l'espressione *lambda* più interna. Quest'ultima prende in input la variabile *c* e calcola la somma dei quadrati di *a*, *b* e *c* (chiudendo lessicalmente le variabili *a* e *b* appartenenti al contesto delle espressioni *lambda* in cui è contenuta).
Supponiamo adesso di voler calcolare la somma dei quadrati dei numeri 3, 4 e 5. L'espressione *lambda* che definisce la funzione *f* prende in input la sola variabile *a*, e questo significa che il metodo *apply* consente di specificare un solo valore da passare in input. Invocando tale metodo e passandogli il valore 3, infatti, si ottiene una nuova funzione *g* (che prende in input due numeri interi *b* e *c* e calcola $a^2 + b^2 + c^2$, con a=3).

```
Function<Integer, Function<Integer, Integer>> g = f.apply(3);
```

L'espressione *lambda* che definisce la funzione *g* è esattamente l'espressione *lambda* intermedia nella definizione iniziale (ovvero quella che prende in input la sola variabile *b* e restituisce un'altra espressione *lambda*, in cui il valore 3 è associato alla variabile *a*). Invocando quindi il metodo *apply* della funzione *g* e passandogli il valore 4 si ottiene una nuova funzione *h* che prende in input un numero intero *c* e calcola $a^2 + b^2 + c^2$, con a=3 e b=4.

```
Function<Integer, Integer> h = g.apply(4);
```

Infine, invocando il metodo *apply* di *h* è possibile calcolare l'effettivo valore della somma dei quadrati dei numeri 3, 4 e 5.

```
System.out.println(h.apply(5)); // visualizza "50"
```

Ovviamente, se le funzioni *g* e *h* non sono necessarie in altre parti del programma, la loro dichiarazione può essere evitata. Si può infatti invocare il metodo *apply* direttamente sulla funzione restituita dall'invocazione dello stesso metodo *apply* su di un'altra funzione.

```
System.out.println(f.apply(3).apply(4).apply(5)); // visualizza "50"
```

I CONCETTI CHIAVE

PROGRAMMAZIONE FUNZIONALE. La programmazione funzionale è un paradigma di programmazione in cui la computazione viene espressa come l'applicazione di una serie di funzioni. Le funzioni sono intese nel senso matematico del termine, in quanto l'output restituito da una funzione dipende sempre e solo dagli argomenti forniti in input alla funzione stessa. In altre parole, dato uno stesso input x, una funzione f restituirà sempre lo stesso valore $f(x)$. Il codice funzionale è infatti indipendente dallo stato del programma in cui è eseguito.

FUNZIONI COME «CITTADINI DI PRIMA CLASSE». Le funzioni possono apparire ovunque all'interno di un programma funzionale, tanto quanto altri «cittadini di prima classe» come i valori (numeri, stringhe, oggetti, ecc.) e le variabili. Una funzione può quindi essere assegnata a una variabile, passata come parametro di input, o restituita in output. Questo consente di definire funzioni di ordine superiore, ovvero funzioni che prendono altre funzioni come parametri di input e/o restituiscono altre funzioni come risultato in output.

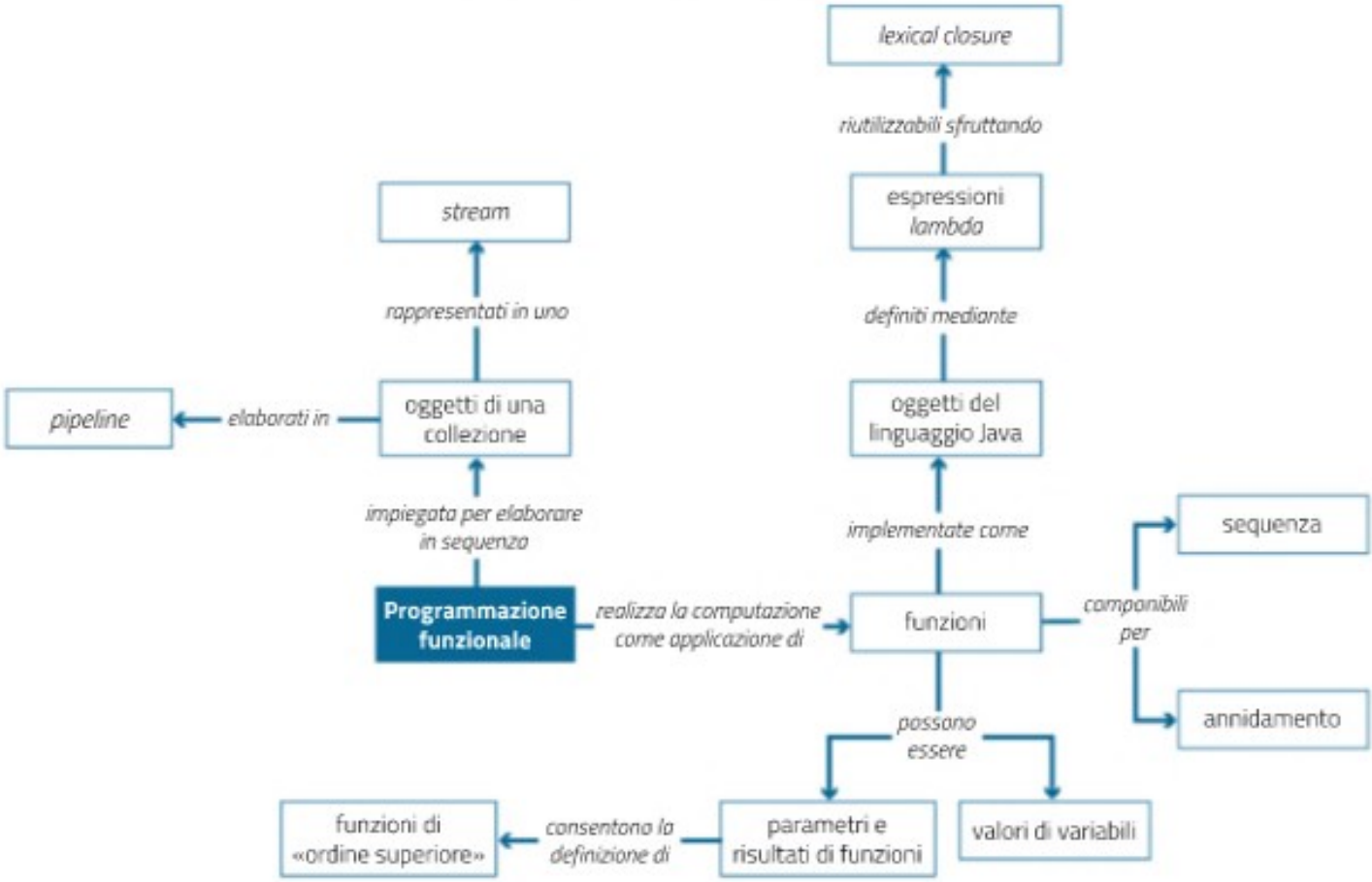
FUNZIONI COME OGGETTI JAVA. Java consente di trattare le funzioni come «cittadini di prima classe», poiché tratta le funzioni come dei normali oggetti Java. Il meccanismo alla base di tutto ciò è quello delle espressioni *lambda*. Un'espressione *lambda* descrive una funzione in modo anonimo, ovvero senza assegnare loro un identificatore. Un'espressione *lambda* è sostanzialmente un blocco di codice preceduto dalla specifica delle variabili che devono essere passate in input a tale codice. Le espressioni *lambda* sono trattate come dei normali oggetti Java che possono essere passati in input e/o restituiti in output da una funzione, o possono essere assegnati a una variabile durante la definizione di una nuova funzione. Il *package java.util.function* mette a disposizione un insieme di interfacce predefinite per la definizione e l'utilizzo di funzioni.

COMPOSIZIONE DI FUNZIONI. Le funzioni implementate in Java possono essere composte mediante i meccanismi di *sequenza* e *annidamento*. Due o più funzioni possono essere composte in sequenza semplicemente utilizzando l'output di una funzione come input della successiva. Il meccanismo di annidamento prevede invece l'utilizzo di una o più funzioni all'interno di un'altra.

FUNZIONI E COLLEZIONI DI OGGETTI. Lo stile funzionale risulta particolarmente efficace quando applicato a collezioni di oggetti. Una collezione di oggetti Java può essere processata sotto forma di *stream*, ovvero accedendo gli oggetti che contiene in sequenza e applicando a tali oggetti una o più funzioni. Per processare una collezione di oggetti sotto forma di *stream*, è sufficiente invocare il metodo *stream* offerto dalla collezione stessa. Ciò risulta nella creazione di un oggetto della classe *Stream<T>*, che consente di processare gli oggetti presenti nella collezione tutti allo stesso modo (ovvero applicandovi le stesse funzioni). Uno *Stream<T>* viene generalmente processato in *pipeline*, ovvero applicandovi una sequenza di operazioni, l'output di ciascuna delle quali è utilizzato come input della successiva. Ciascuna di tali operazioni costituisce uno stadio della *pipeline*, e ciascuno stadio è progettato per applicare opportunamente una funzione agli oggetti contenuti in tale *Stream<T>*. Il primo stadio della *pipeline* si occupa di generare lo *Stream<T>* da processare.

LEXICAL SCOPING* E *LEXICAL CLOSURE. Il *lexical scoping* è il meccanismo che consente di creare una o più variabili in un contesto per poi utilizzarle in contesti annidati all'interno di quello di definizione. Quando un'espressione *lambda* utilizza variabili non locali, ovvero variabili prese dal contesto in cui è definita, si dice che tale espressione *lambda* esegue una *lexical closure* delle variabili non locali. I meccanismi di *lexical scoping* e *lexical closure* risultano particolarmente utili per il riuso di espressioni *lambda*, evitando di dover replicare un'espressione *lambda* quando tale espressione è utilizzata per definire due o più funzioni.

RIPASSA CON LA MAPPA



SPIEGA IN UNA PAGINA

1 **Descrivere il meccanismo delle espressioni *lambda* e spiegare perché costituisce la base del supporto funzionale disponibile in Java.**

2 **Spiegare perché il supporto funzionale disponibile in Java risulta particolarmente utile per processare collezioni di oggetti, indicando esplicitamente il ruolo svolto da *Stream<T>* e *pipeline*.**

QUESITI A RISPOSTA APERTA

3 **Qual è, in generale, la differenza tra *Function<T, R>* e *UnaryOperator<T>*? In quale caso sono invece equivalenti?**

4 **Java consente la definizione di funzioni ricorsive? Altrimenti com'è possibile aggirare il problema?**

5 **Cosa si intende per funzioni di ordine superiore e *lexical scoping*? Come vengono utilizzati per favorire il riuso di espressioni *lambda*?**

TEST A RISPOSTA MULTIPLA

6 **Secondo il paradigma di programmazione funzionale, l'output di una funzione può dipendere:**

A dai parametri forniti in input alla funzione.
B da attributi statici della classe in cui viene utilizzata.
C da attributi d'istanza dell'oggetto in cui viene utilizzata.
D da variabili globali.

7 **Quale delle seguenti affermazioni è falsa?**

A Il supporto funzionale reso disponibile da Java si basa sulle cosiddette espressioni *lambda*.
B L'insieme delle interfacce utilizzabili per definire e utilizzare funzioni in Java è contenuto nel *package java.util.function*.
C Java supporta nativamente la definizione di funzioni ricorsive.
D Java consente di definire e utilizzare una funzione all'interno di un'altra.

8 **Un'espressione *lambda* è:**

A un modo per combinare variabili utilizzando operatori aritmetico-logici.
B un blocco di codice preceduto dalla specifica delle variabili che devono essere passate in input a tale codice.
C un riferimento a un metodo di una classe o di un oggetto.
D la firma di una funzione, ovvero l'indicazione del suo nome, dei parametri che riceve in input e di quelli che restituisce in output.

9 **Java tratta le espressioni *lambda* allo stesso modo:**

A delle classi.
B dei tipi.
C degli oggetti.
D delle interfacce.

10 **Il meccanismo di *method reference* consente di semplificare la scrittura di un'espressione *lambda* che:**

A combina due o più metodi.
B consiste nella sola invocazione di un metodo esistente.
C definisce nuovi metodi o funzioni al suo interno.
D non restituisce mai alcun valore.

11 **Si consideri la funzione *piùUno* che, dato un numero *n*, restituisce il numero *n*+1. Una possibile definizione di *piùUno* è la seguente:**

```
Function<Integer, Integer> piùUno = () ->
(n+1);
```

Tale definizione contiene, però, un errore. Quale?

A La funzione *piùUno* è definita con *Function<Integer, Integer>*, mentre dovrebbe essere definita come *Function<Double, Double>*.
B La funzione *piùUno* è definita con *Function<Integer, Integer>*, mentre dovrebbe essere definita come *Predicate<Integer>*.
C Nell'espressione *lambda* non sono definiti tutti i parametri di input necessari.
D Nell'espressione *lambda* manca l'istruzione `return` con cui viene restituito l'output della funzione *piùUno*.

12 **Quale delle seguenti affermazioni sulla composizione di funzioni è falsa?**

A Due o più funzioni possono essere composte utilizzando i meccanismi di sequenza e/o annidamento.

B Se l'output di una funzione *f* è utilizzato come input di un'altra funzione *g*, si dice che *f* e *g* sono composte in sequenza.

C Ogni volta che una funzione viene utilizzata all'interno di una classe si dice che tale funzione è annidata nella classe.

D Non vi è limite al numero di funzioni che possono essere composte.

13 La classe *Stream<T>* consente di:

A processare gli oggetti presenti nella collezione tutti allo stesso modo.

B accedere singolarmente agli oggetti contenuti in una collezione e processare ciascuno di essi separatamente.

C accedere direttamente agli oggetti contenuti in una collezione e modificarli arbitrariamente.

D processare gli oggetti contenuti in una *pipeline*.

14 Si considerino le seguenti affermazioni riguardanti una *pipeline* utilizzata per processare uno *Stream<T>*. Quale di tali affermazioni è vera?

A Una *pipeline* ha sempre un solo stadio iniziale e un solo stadio finale.

B Una *pipeline* può avere più stadi iniziali.

C Una *pipeline* può avere più stadi finali.

D Una *pipeline* non può contenere più di 10 stadi intermedi.

15 Si consideri la sequenza di stadi che forma una *pipeline*. Quale delle seguenti affermazioni è falsa?

A Uno stadio *filter* può essere posizionato dopo uno stadio *map*.

B Uno stadio *map* può essere posizionato dopo uno stadio *filter*.

C Uno stadio *forEach* può essere posizionato dopo uno stadio *filter*.

D Uno stadio *filter* può essere posizionato dopo uno stadio *forEach*.

16 Il meccanismo di *lexical scoping* consente di:

A definire funzioni di ordine superiore.

B definire funzioni anonime.

C dichiarare una variabile in un'espressione *lambda* e poi utilizzarla nelle espressioni *lambda* in essa annidate.

D utilizzare le funzioni come «cittadini di prima classe».

17 Un'espressione *lambda* esegue una *lexical closure* quando:

A utilizza uno o più dei parametri ricevuti in input.

B utilizza variabili non locali.

C definisce una nuova variabile.

D utilizza i simboli «{« e »}».

18 Si consideri l'esecuzione del seguente metodo *main*:

```
public static void main(String[] args) {
      Function<Integer, Integer> doppio = (n) -> (3*n);
      Function<Integer, Integer> piùUno = (n) -> (n+1);

      Function<Integer, Integer> f1 = doppio.andThen(piùUno);
      Function<Integer, Integer> f2 = piùUno.andThen(doppio);

      if (f1.apply(3) == f2.apply(3))
          System.out.println("Stesso risultato.");
      else
          System.out.println("Risultati diversi.");
}
```

Quale delle seguenti affermazioni è vera?

A Il metodo *main* visualizza la stringa "Stesso risultato".

B Il metodo *main* visualizza la stringa "Risultati diversi".

C Il metodo *main* contiene errori e non potrà mai essere eseguito.

D Il metodo *main* non visualizza nulla.

19 Si consideri il problema del riuso delle espressioni *lambda*. Quale dei seguenti meccanismi non contribuisce a tale riuso?

A Ricorsione.
B Funzioni di ordine superiore.
C *Lexical scoping*.
D *Lexical closure*.

PROBLEMI

20 Codificare in Java una *BiFunction<Integer, Integer, Integer>* che implementa la funzione $p(x, y) = (x^2 - y^2)^3$. La codifica di p deve prevedere un'opportuna composizione delle seguenti funzioni:

- $f(x) = x^2$;
- $g(x, y) = x - y$;
- $h(x) = x^3$.

21 L'algoritmo di Euclide consente di calcolare ricorsivamente il massimo comun divisore tra due numeri n ed m. Tale algoritmo consente infatti di calcolare il valore di $MCD(n, m)$ come indicato di seguito:

- $MCD(n, m) = n$, se $m = 0$;
- $MCD(n, m) = MCD(n, m)$, se $m > n$;
- $MCD(n, m) = MCD(n - m, m)$, altrimenti.

Codificare una funzione *mcd* che implementi il calcolo del massimo comun divisore tra due numeri utilizzando l'algoritmo di Euclide. La funzione *mcd* deve essere implementata utilizzando la metodologia per lo sviluppo di *Function<T, U>* ricorsive. [Suggerimento: poiché tale metodologia consente di costruire funzioni ricorsive che ricevano un solo parametro in input, utilizzare un'opportuna struttura dati per fornire una coppia di interi come parametro di input alla funzione implementata.]

CASI PRATICI

22 A SCUOLA DI LAVORO Un negoziante necessita di un'applicazione Java che consenta di memorizzare e analizzare gli incassi e le spese del suo negozio. In particolare, il negoziante richiede che l'applicazione fornisca le seguenti funzionalità:

A. aggiunta di una nuova operazione, specificandone l'importo (positivo per un incasso o negativo per una spesa), la motivazione e la data;
B. visualizzazione di tutte le operazioni effettuate;
C. verifica della presenza di operazioni effettuate in una precisa data;
D. calcolo delle spese effettuate in un determinato intervallo di tempo;
E. calcolo del totale delle spese determinate dalla stessa motivazione;
F. calcolo degli incassi relativi a un determinato intervallo di tempo;
G. calcolo del totale delle spese determinate dalla stessa motivazione;
H. calcolo del bilancio complessivo.

Progettare e sviluppare le classi necessarie a implementare l'applicazione richiesta dal negoziante, utilizzando opportunamente il supporto funzionale offerto da Java. Realizzare inoltre un metodo *main* che consenta di testare quanto implementato.

23 L'applicazione *GPShare* consente ai propri utenti di utilizzare il proprio smartphone per condividere le proprie coordinate GPS per fini statistici. L'insieme di tutte le coordinate registrate viene memorizzato in un file le cui righe corrispondono ciascuna a una diversa condivisione (memorizzandone la latitudine, la longitudine e il nome dell'utente che ha condiviso la posizione). Progettare e implementare delle classi Java che consentano di analizzare un file analogo a quello prodotto da *GPShare*. Tali classi devono consentire quanto segue:

A. caricamento della collezione di registrazioni da file;
B. conteggio del numero di condivisioni riguardanti una specifica area geografica, indicata in termini di longitudine minima/massima e latitudine minima/massima;
C. calcolo dell'area geografica coperta complessivamente, in termini di longitudine minima/massima e latitudine minima/massima;
D. calcolo dell'area geografica coperta da un determinato utente, in termini di longitudine minima/massima e latitudine minima/massima;
E. determinazione dell'utente che si è recato più vicino a una particolare coppia di coordinate.

Le funzionalità elencate devono essere implementate sfruttando opportunamente il supporto funzionale disponibile in Java. Realizzare inoltre un metodo *main* che consenta di testare le classi Java implementate.

24 Una scuola richiede lo sviluppo di un registro elettronico per memorizzare i voti dei propri alunni. Ciascuna registrazione deve indicare la matricola dell'alunno, il voto assegnatogli e la materia corrispondente. Inoltre, il sistema deve consentire quanto segue:

A. registrazione di un voto;
B. visualizzazione dei voti assegnati a un alunno;
C. calcolo della media complessiva di un alunno;
D. calcolo della media di un alunno in una materia;
E. determinazione dell'insieme di alunni che hanno una media complessiva insufficiente;
F. determinazione dell'insieme di alunni che hanno una media insufficiente in una materia;
G. determinazione dell'insieme di alunni che non hanno voti insufficienti.

Progettare e implementare le classi Java che realizzino le funzionalità elencate, utilizzando opportunamente il supporto funzionale disponibile in Java. Realizzare inoltre un metodo *main* che consenta di testare interattivamente quanto implementato.

25 Progettare e implementare un *package my.polygon*, il quale deve contenere l'insieme di classi Java necessarie a rappresentare e manipolare poligoni geometrici nel piano cartesiano. Tale *package* deve rendere disponibili le seguenti funzionalità:

A. creazione di un poligono, data la lista ordinata dei suoi vertici;
B. visualizzazione dei vertici che compongono un poligono;
C. calcolo del perimetro di un poligono;
D. determinazione del/i quadrante/i del piano cartesiano toccato/i da un poligono;
E. traslazione di un poligono, dati i valori di spostamento lungo l'asse *x* e lungo l'asse *y*.

Tali funzionalità devono essere realizzate utilizzando opportunamente il supporto funzionale disponibile in Java. Realizzare inoltre un metodo *main* che testi le classi Java contenute nel *package* implementato.

26 Un circolo privato richiede una semplice applicazione in grado di memorizzare e analizzare l'insieme dei soci iscritti al circolo, dei quali sono noti nome, cognome, età e sesso. A tale scopo, l'applicazione deve fornire le seguenti funzionalità:

A. caricamento dell'insieme dei soci da file;
B. aggiunta, modifica e rimozione di un socio;
C. incremento dell'età di tutti i soci registrati;
D. calcolo dell'età media dei soci;
E. calcolo dell'età media dei soci di sesso maschile e di quella dei soci di sesso femminile;
F. calcolo della distribuzione percentuale dei soci in base al sesso;
G. esportazione su file dell'insieme dei soci registrati.

Progettare e implementare le classi Java che realizzino le funzionalità elencate, utilizzando opportunamente il supporto funzionale disponibile in Java. Realizzare inoltre un metodo *main* che testi quanto implementato.

A8

Graphical User Interface con JavaFX

Le applicazioni realizzate nei capitoli precedenti hanno un'interfaccia utente testuale e, come tali, risultano utilizzabili solo da esperti in grado di utilizzare questo tipo di interfaccia.

La maggior parte delle applicazioni moderne dispone invece di un'**interfaccia grafica** o **GUI** (*Graphical User Interface*) grazie alla quale è possibile mostrare informazioni all'utente utilizzando oggetti grafici interattivi contenuti in una o più «finestre»; inoltre, una GUI permette modalità di input diverse dalla sola tastiera, per esempio: mouse, touchscreen, joystick, ecc.

ESEMPIO

La seguente applicazione Java permette l'inserimento dei dati anagrafici dei dipendenti di un'azienda utilizzando un'interfaccia testuale:

```
package com.company;

import java.util.ArrayList;
import java.util.Scanner;

/* classe che rappresenta un dipendente */
class Dipendente {
  private String nome;
  private String cognome;
  private String codiceFiscale;

/* costruttore */
public Dipendente(String nome, String cognome, String codiceFiscale) {
  this.nome = nome;
  this.cognome = cognome;
  this.codiceFiscale = codiceFiscale;
}

/* costruttore di copia */
public Dipendente(Dipendente dipendente) {
  this.nome = dipendente.getNome();
  this.cognome = dipendente.getCognome();
  this.codiceFiscale = dipendente.getCodiceFiscale();
}

public String getNome() {
  return nome;
}

public String getCognome() {
  return cognome;
}
```

```
public String getCodiceFiscale() {
  return codiceFiscale;
}

@Override
public boolean equals(Object altro) {
  if (altro == null)
    return false;
  if (altro == this)
    return true;
  if (!(altro instanceof Dipendente))
    return false;
  Dipendente altroDipendente = (Dipendente) altro;
  return altroDipendente.getCodiceFiscale()
         .equalsIgnoreCase(this.getCodiceFiscale());
}
}

/* classe che gestisce una collezione di dipendenti */
public class GestionaleDipendenti {

  private ArrayList<Dipendente> dipendenti;

  /* costruttore */
  public GestionaleDipendenti() {
    dipendenti = new ArrayList<>();
  }

  /* aggiunge un nuovo dipendente */
  public void nuovoDipendente(Dipendente dipendente) {
    dipendenti.add(new Dipendente(dipendente));
  }

  /* elimina un dipendente esistente */
  public void eliminaDipendente(Dipendente dipendente) {
    dipendenti.remove(dipendente);
  }

  /* restituisce numero dipendenti */
  public int numeroDipendenti() {
    return dipendenti.size();
  }

  /* restituisce l'elenco dei dipendenti */
  public Dipendente[] elencoDipendenti() {
    Dipendente[] array = new Dipendente[dipendenti.size()];

    return dipendenti.toArray(array);
  }

  /* main: permette l'inserimento dei dati utilizzando la tastiera */
  public static void main(String[] args) {
    Scanner reader = new Scanner(System.in);
    GestionaleDipendenti gestionale = new GestionaleDipendenti();

    while(true) {
      System.out.println("Nome dipendente:");
      String nome = reader.next();
      System.out.println("Cognome dipendente:");
      String cognome = reader.next();
```

```
        System.out.println("Codice fiscale:");
        String codiceFiscale = reader.next();
        Dipendente dipendente = new Dipendente(nome,
                                               cognome,
                                               codiceFiscale);
        gestionale.nuovoDipendente(dipendente);
        System.out.println("[Dipendente aggiunto]");
        System.out.println("[Totale dipendenti: " +
                    gestionale.numeroDipendenti() + "]");
    }
  }
}
```

Sebbene funzionante, un'applicazione con interfaccia testuale è fortemente limitata in quanto a interazione con l'utente:

```
Nome dipendente:
> Mario
Cognome dipendente:
> Rossi
Codice fiscale:
> MRRSS123456789
[Dipendete aggiunto]
[Totale dipendenti: 1]
...
...
...
```

Non è infatti possibile compilare più campi allo stesso tempo, né controllare i valori inseriti prima di confermarli o ricevere feedback in tempo reale mentre si riempiono i campi. Se la stessa applicazione viene dotata di una GUI si ottiene un programma maggiormente *user friendly*, ovvero più facile da utilizzare (FIGURA **1**).

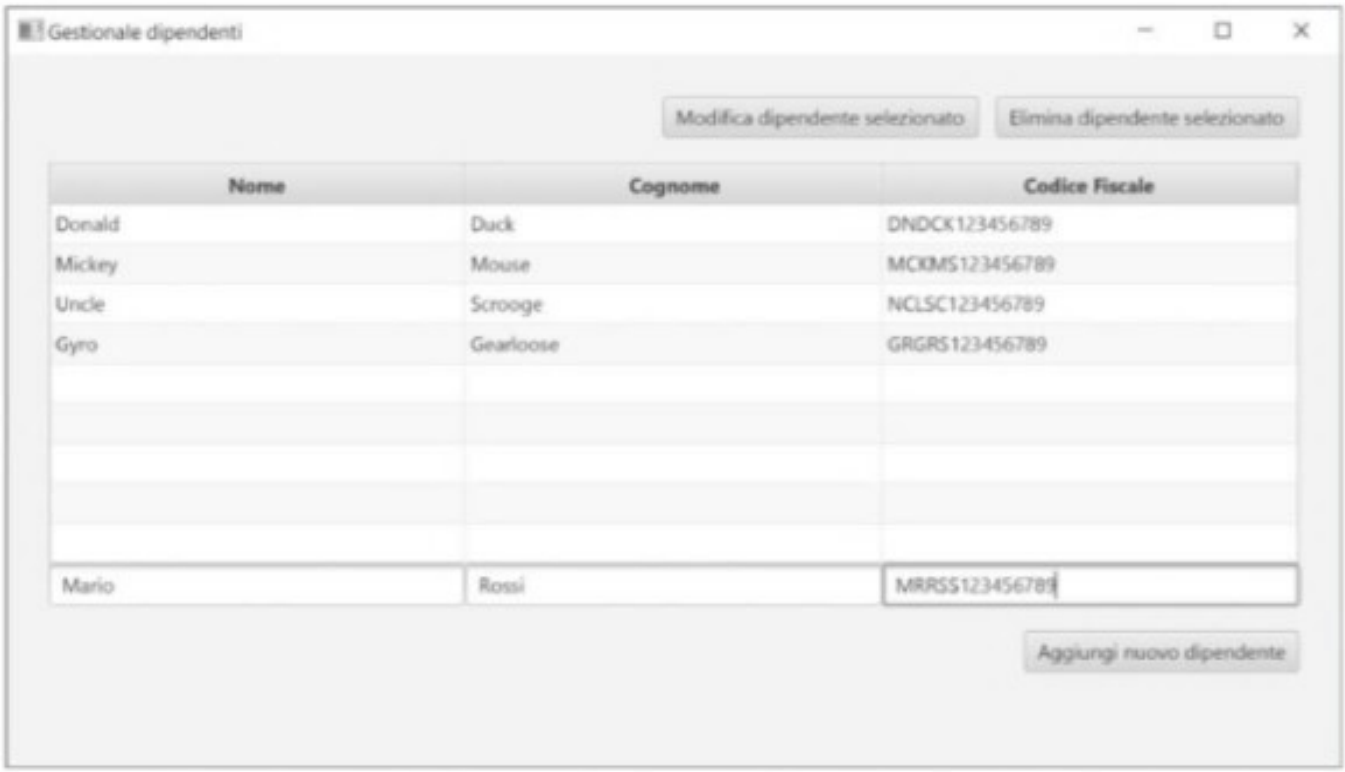


FIGURA **1**

1 Introduzione a JavaFX

JavaFX è un insieme di librerie e componenti standard per la realizzazione di interfacce grafiche per applicazioni Java. È incluso all'interno della distribuzione del linguaggio Java e, di conseguenza, un'applicazione JavaFX, a differenza di altri *framework* per la realizzazione di GUI, non necessita di librerie o componenti aggiuntivi per la propria esecuzione su un sistema su cui è installato il supporto Java.

1.1 *Pattern Model-View-Controller*

Analogamente ad altri *framework* per la realizzazione di GUI, JavaFX rende disponibili strumenti che facilitano l'implementazione del *pattern* architetturale denominato **MVC** (*Model-View-Controller*). Il *pattern* MVC prevede la scomposizione di un'applicazione grafica in tre tipi di componenti:

- *model*: logica interna, indipendente dall'interfaccia grafica, spesso riferita come *business logic*;
- *view*: interfaccia grafica dell'applicazione;
- *controller*: gestione dell'interazione dell'utente con l'interfaccia grafica.

La FIGURA 2 rappresenta graficamente il *pattern* MVC ed evidenzia come l'interazione dell'utente con il *model* sia mediata da *view* (visualizzazione) e *controller* (interazione):

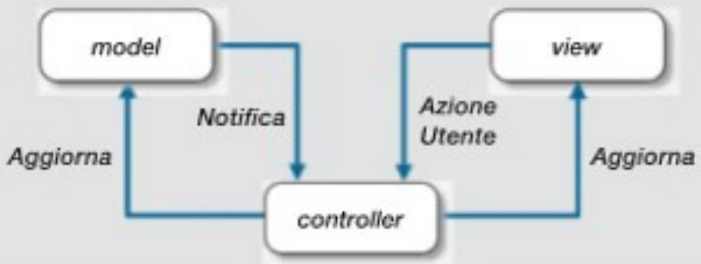


FIGURA 2

Adottare il *pattern* MVC nella realizzazione di un'applicazione contribuisce a rendere più semplici ed efficaci lo sviluppo, la manutenzione e la riusabilità del codice sorgente, in particolare nel caso di applicazioni complesse.

ESEMPIO

L'applicazione di gestione dei dipendenti di un'azienda potrebbe essere pensata come scomposta nei seguenti componenti:

- *Model-Dipendente*: classe che rappresenta un dipendente;
- *Model-GestionaleDipendenti*: classe che contiene la lista dei dipendenti e che espone i metodi per elencare i dipendenti e aggiungerne uno nuovo;
- *View-ElencoDipendenti*: interfaccia grafica contenente l'elenco dei dipendenti, il pulsante per eliminarne uno selezionato e quello per aggiungerne uno nuovo aprendo una nuova *view*;

▶

Storia e caratteristiche di JavaFX

La versione 1.0 di JavaFX è stata rilasciata da Sun Microsystems nel 2008, ma la API (*Application Program Interface*) attuale è stata definita nella versione 2.0 distribuita da Oracle Corporation a partire dal 2011; a partire dalla versione 8 del linguaggio Java rilasciata nel 2014, JavaFX è parte integrante dell'ambiente di sviluppo ed esecuzione e ne assume lo stesso numero di versione. JavaFX è destinato a sostituire il *framework Swing* come strumento per la realizzazione delle GUI per applicazioni Java. Il *look and feel* delle applicazioni JavaFX può essere modificato utilizzando i fogli di stile CSS (*Cascading Style Sheet*) e la GUI di un'applicazione viene normalmente progettata dallo sviluppatore utilizzando un editor grafico, noto come *Scene Builder*, che rende disponibili tutti i componenti grafici necessari per la visualizzazione.

Il *pattern* MVC

Il *pattern* MVC è stato adottato per la prima volta come concetto guida per la realizzazione delle interfacce grafiche negli anni Ottanta del secolo scorso per il linguaggio *Smalltalk* sviluppato nei laboratori del centro di ricerca di Palo Alto della Xerox Corporation. La separazione di responsabilità tra componenti software e il disaccoppiamento tra *business logic* e GUI previsti dal *pattern* sono oggi alla base del progetto di qualsiasi *framework* per la realizzazione di interfacce utente grafiche; in particolare, nelle applicazioni web il *pattern* MVC modella la separazione di funzioni tra client (GUI) e server (*business logic*).

- *View-AggiungiDipendente*: interfaccia grafica per l'inserimento dei dati relativi a un nuovo dipendente comprendente i tre campi compilabili e il pulsante di conferma;
- *Controller-ElencoDipendenti*: gestisce l'interazione dell'utente con la relativa *view*, per esempio apre la finestra d'inserimento di un nuovo dipendente;
- *Controller-AggiungiDipendente*: gestisce l'interazione dell'utente con la relativa *view*, per esempio controlla che tutti i campi del modulo siano stati compilati correttamente.

1.2 Concetti di base

JavaFX adotta l'analogia teatrale per l'attribuzione dei nomi ai propri componenti principali. La finestra che contiene l'applicazione è definita ***stage*** (palco) e al suo interno sono rappresentate una o più ***scene*** (scene). Come le scene di uno spettacolo rappresentano momenti e ambientazioni diverse della storia, in JavaFX ogni *scene* rappresenta una diversa sezione, ovvero una diversa *view*, dell'applicazione.
Ogni *scene* è caratterizzata da un ***node-graph***, ovvero una struttura gerarchica contenente i vari elementi grafici, denominati ***node*** (nodi), che costituiscono la GUI della scena.

La FIGURA 3 riassume quanto detto finora:

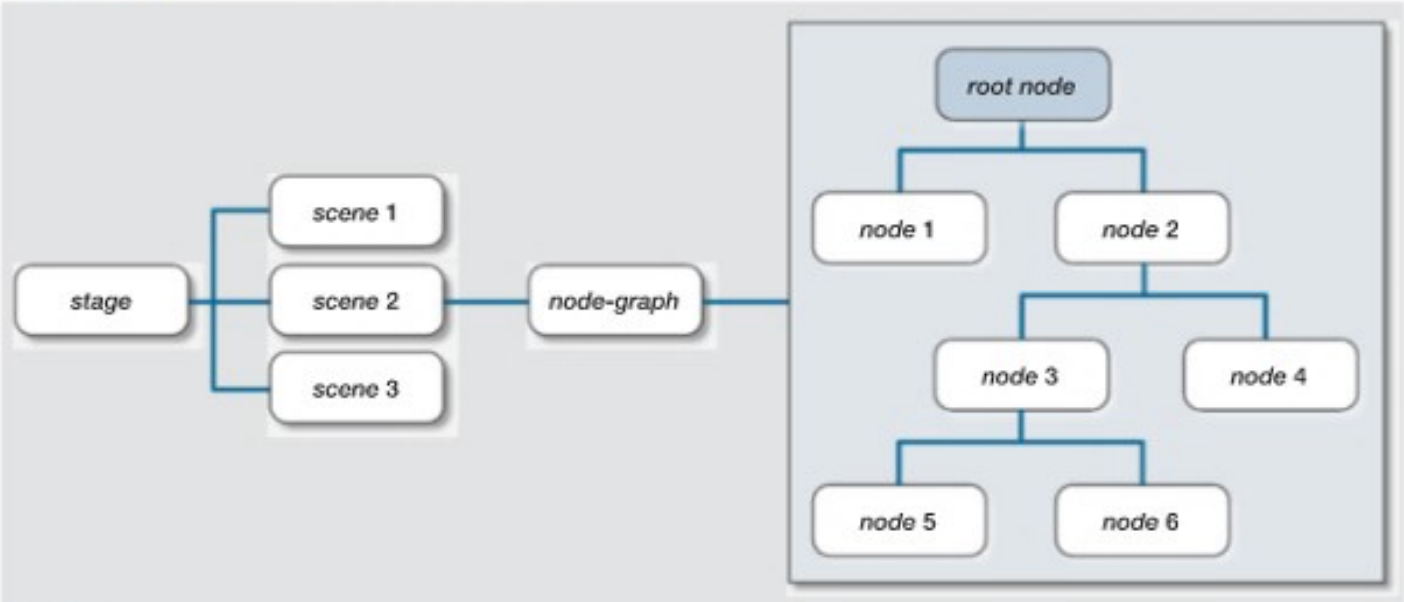


FIGURA 3

JavaFX rende disponibile un gran numero di componenti grafici, ognuno dei quali estende la classe di base *Node* del *package javafx.scene*. Fra i principali tipi di *node* si trovano:

- elementi UI (*User Interface*): elementi dell'interfaccia grafica, per esempio caselle di testo, pulsanti e menu;
- *layout*: strutture che permettono di disporre i componenti grafici sullo schermo; per esempio il *layout VBox* dispone gli elementi uno sotto l'altro in verticale;
- forme geometriche: disponibili sia forme 2D (linea, rettangolo, cerchio, ecc.) sia 3D (cubi, cilindri, ecc.);
- *webview*: permette di incorporare un browser all'interno dell'applicazione.

1.3 FXML

Sebbene sia possibile costruire interfacce grafiche JavaFX utilizzando solo codice Java, è consigliabile utilizzare invece **FXML**, un linguaggio di *markup* basato su XML che è parte integrante del *framework* JavaFX. I file che contengono codice FXML non sono normalmente editati dallo sviluppatore, ma generati mediante ***Scene Builder***, un editor grafico che permette di progettare l'interfaccia grafica di un'applicazione JavaFX.

OSSERVAZIONE La scelta di utilizzare un linguaggio di *markup* per la realizzazione di interfacce grafiche è molto diffusa – si pensi per esempio ad HTML – proprio perché la sintassi di questi linguaggi si presta bene a rappresentare strutture gerarchiche. A dimostrazione di questo si noti come nel codice dell'esempio che segue l'annidamento dei *tag* rispecchia la struttura gerarchica degli elementi grafici che compongono la GUI.

ESEMPIO Il codice FXML che segue comprende un *layout VBox* contenente tre etichette testuali, tre campi di testo e un pulsante:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.VBox?>

<VBox xmlns:fx="http://javafx.com/fxml">
<children>
  <Label text="Nome:"/>
  <TextField fx:id="nomeField"/>
  <Label text="Cognome:"/>
  <TextField fx:id="cognomeField"/>
  <Label text="Codice Fiscale:"/>
  <TextField fx:id="codiceFiscaleField"/>
  <Button text="Aggiungi nuovo dipendente"/>
</children>
</VBox>
```

Ed ecco come appare la semplice interfaccia grafica definita da questo codice FXML; facendo parte di un contenitore *VBox* gli elementi sono disposti uno sotto l'altro (FIGURA 4).

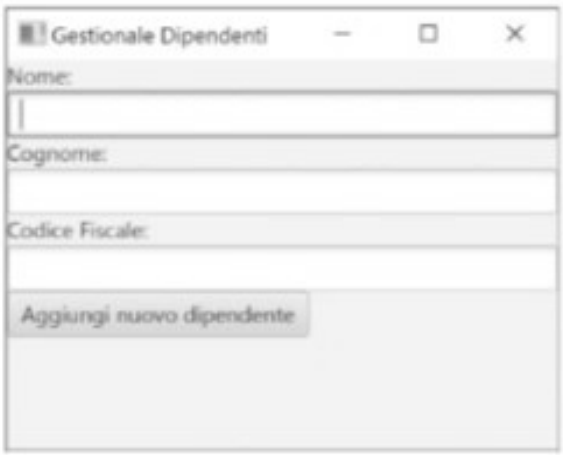


FIGURA 4

Un documento FXML è a sua volta un documento XML, pertanto deve iniziare con la seguente intestazione:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Essendo un documento XML, è possibile anche specificare un *namespace* che sarà utile per validare gli elementi FXML; è convenzione definire un *namespace* denominato *fx* sull'elemento radice del documento:

```
<... xmlns:fx="http://javafx.com/fxml">
```

FXML definisce dei *tag* speciali per l'importazione dei vari tipi di componenti grafici resi disponibili da JavaFX; per esempio, per importare l'elemento *VBox* si utilizza il seguente *tag*:

```
<?import javafx.scene.layout.VBox?>
```

Come vedremo nel seguito, ogni componente grafico di un documento FXML può specificare un *id* che consenta di riferirlo dal codice Java: l'attributo *fx:id* ha esattamente questo scopo.

1.4 *Rendering* di un file FXML e classe *Application*

Il codice Java di un'applicazione deve integrare il codice FXML che ne definisce l'interfaccia grafica. A questo scopo, per realizzare un'applicazione grafica con JavaFX, la sua classe principale dovrà estendere la classe *Application* del *package javafx.application* e sovrascriverne il metodo *start* il cui unico parametro è il riferimento allo *stage* (finestra) dell'applicazione stessa. Nel codice del metodo *start* è possibile caricare un file FXML e creare una nuova *scene* (scena) a partire dal contenuto del file FXML caricato invocando il metodo statico *load* della classe *FXMLLoader* del *package javafx.fxml*. Infine si può mostrare la scena sullo *stage*, che comporta il *rendering* dei componenti grafici definiti nel file FXML all'interno della finestra dell'applicazione invocando il metodo *show* dello *stage* principale.

ESEMPIO Il seguente codice Java mostra il caricamento e la successiva visualizzazione del contenuto di un file FXML:

```
package com.company;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class GestionaleDipendentiApp extends Application {

  @Override
  public void start(Stage primaryStage) throws Exception {
    // imposta il titolo dello stage
    primaryStage.setTitle("Gestionale Dipendenti");
```

```
        // ottiene il riferimento al nodo radice del documento FXML di cui viene
        // specificato il nome del file che lo contiene
        Parent rootNode = FXMLLoader.load(getClass().
                                getResource("nuovoDipendente.fxml"));
        // crea una nuova scena specificandone la larghezza e l'altezza in pixel
        Scene homeScene = new Scene(rootNode, 300, 250);
        // mostra la scena sullo stage
        primaryStage.setScene(homeScene);
        // mostra lo stage
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

L'invocazione concatenata dei metodi *getClass().getResource(...)* ereditati rispettivamente dalle classi *Object* e *Class* consente di ottenere un URL che riferisce la risorsa fornita come argomento, in questo caso il nome del file contenente il codice FXML.
Il metodo *launch* della classe *Application* invocato nel metodo *main* lancia l'esecuzione dell'applicazione invocando indirettamente il metodo *start* sovrascritto.

La TABELLA 1 illustra alcuni metodi della classe *Application*.

TABELLA 1

Metodo	Descrizione
`init()`	Metodo di inizializzazione dell'applicazione JavaFX invocato automaticamente dal *run-time*; se non sovrascritto il metodo è vuoto e non esegue nulla.
`launch(String... args)`	Metodo statico che lancia l'esecuzione di un'applicazione di tipo *stand-alone* a cui sono forniti gli argomenti della riga di comando.
`start(Stage primaryStage)`	*entry-point* dell'applicazione JavaFX: deve essere fornito come argomento lo *stage* principale dell'applicazione; il metodo è astratto e deve essere necessariamente sovrascritto.
`stop()`	Metodo invocato automaticamente dal *run-time* JavaFX al momento della chiusura dell'applicazione; se non sovrascritto il metodo è vuoto e non esegue nulla.

OSSERVAZIONE Il *run-time* JavaFX costruisce un'istanza della classe *Application* e invoca in sequenza i metodi *init* e *start*; al momento della chiusura dell'applicazione invoca il metodo *stop*. Come illustrato nell'esempio precedente la ridefinizione del metodo astratto *start* è obbligatoria e deve prevederne la costruzione e la visualizzazione dello *stage* principale, normalmente caricandone l'aspetto grafico definito in un file FXML; i metodi *init* e *stop* hanno un'implementazione concreta vuota e lo sviluppatore li può eventualmente usare per il codice di inizializzazione e distruzione delle risorse di tipo non grafico utilizzate dall'applicazione.

1.5 Classe *Stage*

La classe ***Stage*** del *package javafx.stage* consente di definire e manipolare la finestra di visualizzazione della GUI di un'applicazione JavaFX.

Nella tabella che segue sono riportati i metodi di maggiore utilità, compresi alcuni ereditati dalla classe *Window* dello stesso *package*.

Metodo	Descrizione
`Stage` `Stage(StageStyle style)`	Costruttore di un'istanza della classe; lo stile può assumere uno dei seguenti valori: *StageStyle.DECORATED*: finestra con sfondo bianco e decorazioni tipiche della piattaforma di esecuzione; *StageStyle.UNDECORATED*: finestra con sfondo bianco, senza le decorazioni tipiche della piattaforma; *StageStyle.TRANSPARENT*: finestra con sfondo trasparente, senza le decorazioni tipiche della piattaforma; *StageStyle.UTILITY*: finestra con sfondo bianco, con un sottoinsieme minimale di decorazioni tipiche della piattaforma.
`centerOnScreen()`	Visualizza la finestra al centro dello schermo.
`close()`	Chiude la finestra dello *stage* (equivale all'invocazione del metodo ereditato *hide*).
`double getMaxHeight()` `double getMaxWidth()`	Restituiscono le rispettive dimensioni massime che la finestra può assumere.
`double getMinHeight()` `double getMinWidth()`	Restituiscono le rispettive dimensioni minime che la finestra può assumere.
`StageStyle getStyle()`	Restituisce lo stile della finestra.
`double getHeight()` `double getWidth()`	Restituiscono le rispettive dimensioni attuali della finestra.
`Scene getScene()`	Restituisce la scena attualmente visualizzata.
`String getTitle()`	Restituisce il titolo della finestra.
`double getX()` `double getY()`	Restituiscono le rispettive posizioni della finestra rispetto allo schermo.
`initModality(Modality modality)`	Imposta la modalità della finestra; può assumere uno dei seguenti valori: *Modality.NONE*: la finestra non blocca nessuna altra finestra; *Modality.WINDOW_MODAL*: la finestra blocca le finestre dell'applicazione da cui dipende; *Modality.APPLICATION_MODAL*:la finestra blocca tutte le altre finestre dell'applicazione.
`setMaxHeight(double h)` `setMaxWidth(double w)`	Impostano le rispettive dimensioni massime che la finestra può assumere.
`setMinHeight(double h)` `setMinWidth(double w)`	Impostano le rispettive dimensioni minime che la finestra può assumere.
`setScene(Scene scene)`	Imposta la scena da visualizzare nella finestra.
`setTitle(String title)`	Imposta il titolo della finestra.
`setHeight(double h)` `setWidth(double w)`	Impostano le rispettive dimensioni attuali della finestra.
`setX(double w)` `setY(double y)`	Impostano le rispettive posizioni della finestra rispetto allo schermo.
`show()`	Visualizza la finestra.
`showAndWait()`	Visualizza la finestra e attende per la chiusura.
`sizeToScene()`	Imposta le dimensioni della finestra a quelle della scena visualizzata.
`toBack()`	Visualizza la finestra in secondo piano (*background*).
`toFront()`	Visualizza la finestra in primo piano (*foreground*).

Risulta ovviamente fondamentale l'invocazione del metodo *setScene* a cui fornire come argomento la scena da visualizzare – invocando successivamente il metodo *show* – costruita a partire dal contenuto di un file FXML: a questo proposito il costruttore di un oggetto di classe *Scene*, oltre al riferimento al nodo radice dello *SceneGraph*, può avere come parametri le dimensioni orizzontale (*width*) e verticale (*height*) della scena stessa e il suo colore di sfondo.

1.6 Classe *controller*

Per assicurare il funzionamento di un'applicazione JavaFX con una GUI che implementi il *pattern* MVC è necessario associare a una scena, costruita a partire dal contenuto di un file FXML, una classe *controller* che gestisca l'interazione fra l'utente e l'interfaccia grafica.
Per indicare la classe *controller* nel file FXML si ricorre all'attributo *fx:controller* dell'elemento radice: il valore da assegnare all'attributo è il nome della classe Java che implementa il *controller*.

ESEMPIO

Per esempio il seguente codice indica la classe *AggiungiDipendenteController* del *package com.company* come *controller* della scena definita dal file FXML:

```
<VBox xmlns:fx=http://javafx.com/fxml
   fx:controller="com.company.AggiungiDipendenteController">
...
</VBox>
```

JavaFX sempifica il lavoro dello sviluppatore di applicazioni Java dotate di GUI gestendo in autonomia i riferimenti fra elementi della *scene* FXML, come, per esempio, un pulsante, o un campo di testo, e gli attributi e i metodi della relativa classe *controller*. In questo modo risulta semplice indicare quale metodo invocare alla pressione di un pulsante da parte dell'utente, o stabilire come accedere al contenuto di un campo di testo.

Nel file FXML è necessario attribuire un identificativo univoco agli elementi della GUI a cui si vuole accedere dal codice della classe *controller*: a questo scopo si usa l'attributo *fx:id* di FXML a cui viene assegnato un valore che è l'identificativo del componente grafico; nel seguente modo, per esempio, si assegna l'identificativo *output* a una casella di testo:

```
<TextField fx:id="output"/>
```

Una volta assegnato un identificativo a un componente grafico della GUI nel file FXML è possibile avere un riferimento allo stesso all'interno della classe *controller* dichiarando una proprietà della classe il cui nome corrisponde all'identificativo stesso; per consentire l'associazione da parte di JavaFX in fase di compilazione è necessario inserire il decoratore **@FXML** prima della dichiarazione della proprietà:

```
// riferimento all'elemento con identificativo "output"
@FXML private TextField output;
```

Le operazioni di inizializzazione del *controller* non devono essere effettuate dal codice del costruttore della classe, ma dal codice di un metodo, denominato *initialize*, la cui dichiarazione deve ovviamente essere preceduta dal decoratore **@FXML**. A differenza del costruttore della classe, che viene eseguito al momento che viene creata l'istanza del *controller*, il metodo *initialize* viene invocato dal *run-time* JavaFX dopo che gli attributi e i metodi etichettati con **@FXML** sono già stati collegati ai rispettivi elementi della GUI.

```
    // event handler per pulsante aggiungiDipendenteButton
    public void handleAggiungiDipendente(MouseEvent event) {
      // codice per aggiungere un dipendente
      ...
      ...
      ...
    }
}
```

Oltre al metodo generico *addEventHandler*, il quale richiede il tipo di evento come primo argomento, JavaFX mette a disposizione dei metodi di convenienza (*convenience method*) che non richiedono di specificare il tipo di evento da catturare in quanto esso è specificato dal nome del metodo:

```
setOnTipoDiEvento(...);
```

ESEMPIO Il seguente codice mostra l'utilizzo di un *convenience method* per la registrazione di un *event handler* da eseguire al click del mouse su di un pulsante.

```
// registrazione event handler con convenience method
aggiungiDipendenteButton.setOnMouseClicked(this::handleAggiungiDipendente);
```

Una tecnica molto utilizzata è quella di registrare gli *event handler* direttamente all'interno del file FXML di definizione della *scene*. Per farlo si utilizzano degli attributi dei nodi il cui nome è molto simile a quello dei relativi *convenience method* e il cui valore è il nome del metodo del *controller* che si desidera registrare come *event handler* preceduto dal simbolo «#»:

```
onTipoDiEvento="#metodoHandler"
```

ESEMPIO Il seguente codice registra il metodo *handleAggiungiDipendente* della classe *controller* come *event handler* di un pulsante per lo stesso tipo di evento degli esempi precedenti:

```
<Button text="Aggiungi nuovo dipendente"
  onMouseClicked="#handleAggiungiDipendente"/>
```

La seguente tabella elenca i tipi di evento più comuni relativi a mouse e tastiera.

Tipo di evento	*convenience method*	Attributo FXML	Descrizione
MouseEvent.MOUSE_PRESSED	*setOnMousePressed*	*onMouseReleased*	Pressione del tasto del mouse.
MouseEvent.MOUSE_RELEASED	*setOnMouseReleased*	*onMouseReleased*	Rilascio del tasto del mouse.
MouseEvent.MOUSE_CLICKED	*setOnMouseClicked*	*onMouseClicked*	Click con il tasto del mouse.
MouseEvent.MOUSE_ENTERED	*setOnMouseEntered*	*onMouseEntered*	Ingresso del cursore del mouse nell'area del componente.
MouseEvent.MOUSE_EXITED	*setOnMouseExited*	*onMouseExited*	Uscita del cursore del mouse dall'area del componente.
KeyEvent.KEY_PRESSED	*setOnKeyPressed*	*onKeyPressed*	Pressione del tasto della tastiera.
KeyEvent.KEY_RELEASED	*setOnKeyReleased*	*onKeyReleased*	Rilascio del tasto della tastiera.
KeyEvent.KEY_TYPED	*setOnKeyTyped*	*onKeyTyped*	Digitazione di un tasto.

La propagazione di un evento lungo la catena di *dispatch* può essere interrotta consumando l'evento. Gli *event filter* e gli *event handler* possono consumare un evento invocando il metodo *consume* dell'oggetto evento. È importante notare che se un *event filter* consuma un evento, questo non raggiungerà mai l'elemento *target* e, di conseguenza, nessun *event handler* sarà invocato.

1.8 Proprietà e *binding*

Oltre a gestire i riferimenti a elementi FXML, JavaFX permette di definire dei *binding* (legami) fra le proprietà della classe *controller* e le proprietà degli elementi grafici di una *scene*. Per esempio, effettuando un *binding* fra una proprietà del *controller* e il valore di un campo di testo della *scene*, il valore inserito dall'utente all'interno del campo sarà automaticamente sincronizzato con la relativa proprietà della classe *controller*.

Per definire una proprietà JavaFX è sufficiente utilizzare una delle classi *wrapper* contenute nel *package javafx.beans.property*. Per esempio per definire una proprietà testuale si utilizza il seguente codice:

```
import javafx.beans.property.StringProperty;
import javafx.beans.property.SimpleStringProperty;

public class ClasseController {
  private StringProperty nome = new SimpleStringProperty("...");
}
```

Oltre a definire la proprietà, è necessario definire i relativi metodi *getter* e *setter* più un terzo metodo che restituisce la proprietà stessa:

```
import javafx.beans.property.StringProperty;
import javafx.beans.property.SimpleStringProperty;

public class ClasseController {
  /* proprietà */
  private StringProperty nome = new SimpleStringProperty("...");

  /* getter e setter */
  public String getNome() { return nome.get(); }

  public void setNome(String nome) { this.nome.set(nome); }

  public StringProperty nomeProperty() { return nome; }
}
```

Come si evince dal codice, le classi *wrapper* di JavaFX espongono i metodi *get* e *set* che permettono di impostare o restituire il valore di una proprietà.

La tabella alla pagina seguente elenca le principali classi presenti nel *package javafx.beans.property*.

Classe	Classe astratta di derivazione	Descrizione
SimpleBooleanProperty	*BooleanProperty*	Proprietà booleana.
ReadOnlyBooleanWrapper	*BooleanProperty*	Proprietà booleana *read-only*.
SimpleIntegerProperty	*IntegerProperty*	Proprietà di tipo intero.
ReadOnlyIntegerWrapper	*IntegerProperty*	Proprietà di tipo intero *read-only*.
SimpleDoubleProperty	*DoubleProperty*	Proprietà di tipo *Double*.
ReadOnlyDoubleWrapper	*DoubleProperty*	Proprietà di tipo *Double read-only*.
SimpleStringProperty	*StringProperty*	Proprietà di tipo *String*.
ReadOnlyStringWrapper	*StringProperty*	Proprietà di tipo *String read-only*.

Per definire un *binding* unidirezionale fra una proprietà del *controller* e un attributo di un elemento FXML si utilizza la seguente sintassi:

```
${controller.proprietà}
```

ESEMPIO Per visualizzare all'interno di una *label* della GUI il contenuto della proprietà *nome* del *controller* si utilizza il codice FXML che segue; come conseguenza del legame definito, quando la proprietà viene modificata all'interno del codice del *controller*, il contenuto visualizzato della *label* viene aggiornato automaticamente.

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>

<VBox xmlns:fx="http://javafx.com/fxml"
  fx:controller="com.company.ClasseController">
<children>
  <Label text="${controller.nome}"/>
</children>
</VBox>
```

Le proprietà JavaFX permettono anche la notifica della variazione del loro valore. È possibile infatti registrare degli *event handler* che sono eseguiti quando il valore di una proprietà cambia: questo risulta particolarmente utile visto che la maggior parte delle proprietà degli elementi grafici di JavaFX sono di solito implementate come proprietà JavaFX. Si può, per esempio, registrare un *handler* sulla proprietà *textProperty* di un campo di testo che verrà invocato a ogni cambiamento del contenuto del campo.

ESEMPIO Il codice che segue mostra come registrare un *handler* che viene eseguito ogniqualvolta il valore di un campo di testo viene modificato dall'utente (il codice dello *handler* trasforma il contenuto del campo in formato maiuscolo):

```
import javafx.fxml.FXML;
import javafx.scene.control.TextField;

public class ClasseController {
  /* riferimenti FXML */
  @FXML private TextField field;

  @FXML
  public void initialize() {
```

```
        // registra un event handler che viene eseguito
        // a ogni cambiamento del valore del campo field
        field.textProperty().addListener(
           (observable, oldValue, newValue) -> {
           field.textProperty().setValue(newValue.toUpperCase());
           }
        );
     }
  }
```

È possibile definire un *binding* bidirezionale, ovvero che consente di modificare una proprietà sia dal codice del *controller* sia mediante l'interazione dell'utente, con un componente grafico della GUI definito in una *scene* FXML. Questo comportamento risulta particolarmente utile nel caso di un campo di testo: con un *binding* bidirezionale è infatti possibile fare in modo che una proprietà del *controller* venga aggiornata quando l'utente cambia il valore del campo. A differenza dei *binding* unidirezionali, un *binding* bidirezionale non può essere definito all'interno del file FXML ma solo all'interno del *controller*.

Il seguente codice mostra come definire un *binding* bidirezionale fra una proprietà del *controller* e il valore di un campo di testo della *scene*:

```
import javafx.fxml.FXML;
import javafx.beans.property.StringProperty;
import javafx.beans.property.SimpleStringProperty;

public class ClasseController {
  /* proprietà */
  private StringProperty nome = new SimpleStringProperty("...");

  /* riferimenti FXML */
  @FXML private TextField field;

  /* getters e setters */
  public String getNome() { return nome.get(); }

  public void setNome(String nome) { this.nome.set(nome); }

  public StringProperty nomeProperty() { return nome; }

  @FXML
  public void initialize() {
    // binding bidirezionale
    this.field.textProperty().bindBidirectional(this.nome);
  }
}
```

OSSERVAZIONE I *binding* bidirezionali aiutano a ridurre la dipendenza fra *controller* e *scene* FXML: invece di registrare *handler* di eventi su specifici elementi della scena, è infatti possibile definire dei *binding* bidirezionali su alcune proprietà del *controller* e scrivere del codice che reagisce ai cambiamenti dei valori di tali proprietà.

ESEMPIO

Nel seguito viene effettuato un *binding* biderezionale fra i valori di tre campi di testo (*campoNome*, *campoCognome* e *campoCodiceFiscale*) della *scene* FXML e tre proprietà della classe *controller* denominate *nome*, *cognome* e *codiceFiscale*. Il *controller* definisce anche una proprietà *messaggioValidazione*, utilizzata per visualizzare i risultati della validazione dell'input: il messaggio risultante viene visualizzato effettuando un *binding* unidirezionale fra una *label* di testo della *scene* FXML e la relativa proprietà del *controller*.

Classe principale:

```
package com.company;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class GestionaleDipendentiApp extends Application {
  @Override
  public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("Gestionale Dipendenti");
    Parent rootNode =
    FXMLLoader.load(getClass().getResource("aggiungiDipendente.fxml"));
    Scene homeScene = new Scene(rootNode, 300, 250);
    primaryStage.setScene(homeScene);
    primaryStage.show();
  }

  public static void main(String[] args) {
    launch(args);
  }
}
```

Classe *controller*:

```
package com.company;

import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.fxml.FXML;
import javafx.scene.control.TextField;
import javafx.scene.input.MouseEvent;

public class AggiungiDipendenteController {
  /* proprietà */
  private StringProperty nome = new SimpleStringProperty();
  private StringProperty cognome = new SimpleStringProperty();
  private StringProperty codiceFiscale = new SimpleStringProperty();
  private StringProperty messaggioValidazione = new SimpleStringProperty();

  /* riferimenti FXML */
  @FXML private TextField campoNome;
  @FXML private TextField campoCognome;
  @FXML private TextField campoCodiceFiscale;

  @FXML
  public void initialize() {
    // binding
    this.campoNome.textProperty().bindBidirectional(this.nome);
    this.campoCognome.textProperty().bindBidirectional(this.cognome);
    this.campoCodiceFiscale.textProperty().bindBidirectional(this.codiceFiscale);
  }
```

```java
    /* getter e setter */
    public String getNome() { return nome.get(); }
    public StringProperty nomeProperty() { return nome; }
    public void setNome(String nome) { this.nome.set(nome); }

    public String getCognome() { return cognome.get(); }
    public StringProperty cognomeProperty() { return cognome; }
    public void setCognome(String cognome) { this.cognome.set(cognome); }

    public String getCodiceFiscale() { return codiceFiscale.get(); }
    public StringProperty codiceFiscaleProperty() { return codiceFiscale; }
    public void setCodiceFiscale(String codiceFiscale) {
      this.codiceFiscale.set(codiceFiscale);
    }

    public String getMessaggioValidazione() { return messaggioValidazione.get(); }
    public StringProperty messaggioValidazioneProperty() {
      return messaggioValidazione;
    }
    public void setMessaggioValidazione(String msg) {
      this.messaggioValidazione.set(msg);
    }

    /* event handler */
    public void handleAggiungiDipendente(MouseEvent event) {
      if (validaForm()) {
      // codice per aggiungere un nuovo dipendente
      // i valori da usare sono resi disponibili invocando
      // getNome(), getCognome() e getCodiceFiscale()
      ...
      ...
      ...
      }
    }

    private boolean validaForm() {
      boolean valido = true;
      String msg = "";

      if (getNome() == null || getNome().isEmpty()) {
        msg += "Inserisci nome\n";
        valido = false;
      }
      if (getCognome() == null || getCognome().isEmpty()) {
        msg += "Inserisci cognome\n";
        valido = false;
      }
      if (getCodiceFiscale() == null || getCodiceFiscale().isEmpty()) {
        msg += "Inserisci codice fiscale\n";
        valido = false;
      }
      setMessaggioValidazione(msg);
      return valido;
    }

  }
```

Scene FXML:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.control.Button?>

<VBox xmlns:fx="http://javafx.com/fxml"
  fx:controller="com.company.AggiungiDipendenteController">
<children>
  <Label text="Nome:"/>
  <TextField fx:id="campoNome"/>
  <Label text="Cognome:"/>
  <TextField fx:id="campoCognome"/>
  <Label text="Codice Fiscale:"/>
  <TextField fx:id="campoCodiceFiscale"/>
  <Button text="Aggiungi nuovo dipendente"
    onMouseClicked="#handleAggiungiDipendente"/>
  <Label text="${controller.messaggioValidazione}"/>
</children>
</VBox>
```

La FIGURA 7 illustra un caso d'uso del *form* per l'aggiunta di un nuovo dipendente.

FIGURA 7

2 Componenti grafici JavaFX

Ogni singola scena di un'applicazione JavaFX è definita da un file FXML; il file viene normalmente generato utilizzando uno strumento integrato nei principali IDE per il linguaggio di programmazione Java, ***Scene Builder***, che permette di comporre visualmente il *layout* grafico di una scena inserendovi con l'uso del mouse i componenti grafici resi disponibili da JavaFX e presenti sotto forma di icone nella libreria dei componenti. Una volta inseriti nel *layout* della scena i componenti possono essere singolarmente selezionati e configurati impostandone le relative proprietà che usualmente corrispondono agli attributi dell'elemento corrispondente nel file FXML generato.

La FIGURA 8 mostra lo strumento *Scene Builder* in esecuzione.

FIGURA 8

La libreria di *Scene Builder* comprende alcuni componenti di base che possono essere integrati importando altre librerie di componenti JavaFX:

- *container*: componenti grafici che possono, in modo dipendente dalle proprie caratteristiche, comprendere al proprio interno altri componenti grafici;
- *control*: componenti grafici destinati all'interazione diretta con l'utente: *label* e campi di testo, pulsanti, *check-box* e *radio button*, *choice-box* e *combo-box*, tabelle e relative colonne;
- *menu*: componenti grafici per la realizzazione di un menu dei comandi;
- *shape*: componenti grafici per la visualizzazione di figure geometriche in 2 e 3 dimensioni: linee, archi, cerchi, rettangoli, poligoni, curve, cilindri, sfere, parallelepipedi, ecc.;
- *chart*: grafici di vario tipo: cartesiani, lineari, istogrammi, a torta, ecc.

In generale la configurazione iniziale dei componenti grafici JavaFX può essere effettuata utilizzando *Scene Builder*, oppure invocando nel codice di inizializzazione del *controller* i metodi di configurazione degli oggetti Java che corrispondono agli elementi della scena: nel primo caso la configurazione sarà modellata dal codice del file FXML generato dallo strumento.

Nel seguito sono presentati alcuni *container* e *control* utili per la realizzazione di applicazioni con una GUI elementare.

2.1 Contenitori

Un componente di tipo **contenitore** comprende al proprio interno altri componenti, compresi altri contenitori: questo permette una strutturazione gerarchica della disposizione dei componenti dell'interfaccia grafica che si riflette nella struttura ad albero del codice FXML generato.

Ciascun contenitore ha un modo diverso di ridisporre, visualizzare o nascondere il proprio contenuto quando la finestra, o un altro contenitore, che lo contiene viene ridimensionato: questo comportamento può essere parzialmente impostato mediante le proprietà di configurazione esposte.

2.1.1 *Border pane*

Un contenitore di tipo *border pane* (classe *BorderPane* del *package javafx.scene.layout*) consente di disporre i componenti grafici in 5 diverse zone: *top*, *bottom*, *left*, *right* e *center*, ciascuna delle quali può essere presente o meno (FIGURA 9).

FIGURA 9

Il contenitore *border pane* riflette la classica organizzazione di un'interfaccia grafica: menu e *toolbar* in alto, informazioni di stato in basso, gli strumenti di navigazione a sinistra e lo spazio per le informazioni aggiuntive a destra di un'area di lavoro centrale.

ESEMPIO

Il codice FXML di una scena realizzata con un contenitore *border pane* ha come elementi innestati nell'elemento che rappresenta il contenitore gli elementi che corrispondono alle varie zone:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.BorderPane?>

<BorderPane maxHeight="800" maxWidth="1200" minHeight="200" minWidth="300"
    prefHeight="400.0" prefWidth="600.0"
    xmlns:fx="http://javafx.com/fxml">
  <center>
   ...
   ...
   ...
  </center>
  <bottom>
   ...
   ...
   ...
  </bottom>
```

```
    <top>
     ...
     ...
     ...
    </top>
    <left>
     ...
     ...
     ...
    </left>
    <right>
     ...
     ...
     ...
    </right>
  </BorderPane>
```

Si noti come gli attributi dell'elemento *BorderPane* riflettano le proprietà del contenitore impostate, in questo caso: altezza e larghezza massime, minime e di default.

2.1.2 *V-box* e *H-box*

I contenitori di tipo *V-box* (classe *VBox* del *package javafx.scene.layout*) e *H-box* (classe *HBox* del *package javafx.scene.layout*) consentono rispettivamente di organizzare i componenti che contengono in un allineamento verticale o orizzontale (FIGURA **10**).

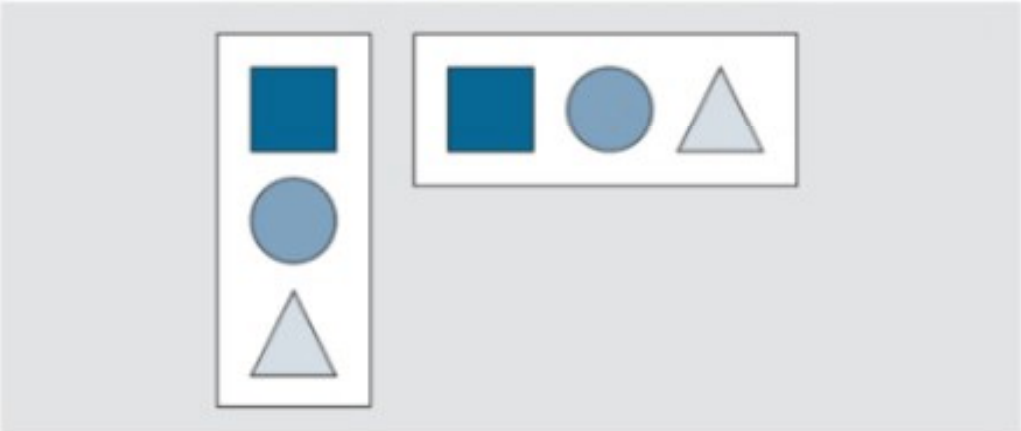


FIGURA **10**

Contenitori di questo tipo sono spesso utilizzati per organizzare i componenti delle zone *top/bottom* (*H-box*) e *left/right* (*V-box*) di un contenitore di tipo *border pane*. Tra le proprietà configurabili due risultano importanti:

- *padding* che permette di stabilire la distanza degli elementi contenuti dai bordi superiore (*top*), destro (*right*), inferiore (*bottom*) e sinistro (*left*) del contenitore;
- *spacing* che consente di impostare la distanza tra gli elementi contenuti.

Queste proprietà, così come tutte le altre, possono essere impostate staticamente nel file FXML, oppure dinamicamente da codice invocando rispettivamente i metodi *setPadding* e *setSpacing*:

```
@FXML VBox box;
...
```

```
...
...
box.setSpacing(10);
box.setPadding(new Insets(20, 20, 20, 20));
...
```

L'oggetto *Insets*, fornito come argomento al metodo *setPadding*, è una quadrupla di valori numerici che nell'ordine rappresentano il *padding* dal bordo superiore, destro, inferiore e sinistro.

2.1.3 *Grid pane*

Un contenitore di tipo *grid pane* (classe *GridPane* del *package javafx.scene.layout*) consente di definire una griglia organizzata in righe e colonne nelle cui celle è possibile disporre i componenti:

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)

Nel file FXML un componente grafico viene disposto in una specifica cella della griglia – che si adatta automaticamente alle dimensioni dei componenti contenuti se non altrimenti specificato – specificandone le coordinate come valori degli attributi *GridPane.rowIndex* e *GridPane.columnIndex*. Il numero di righe e di colonne è determinato dal posizionamento nelle celle dei componenti grafici, mentre la distanza orizzontale tra righe e verticale tra colonne è data dal valore degli attributi *hgap* e *vgap*.

ESEMPIO Il codice FXML di una scena realizzata con un contenitore *grid pane* ha come elementi «figli» i componenti inseriti nella griglia specificando per ciascuno gli indici della cella:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.TextField?>
<?import javafx.scene.layout.Label?>
<?import javafx.scene.layout.Button?>

<GridPane xmlns:fx="http://javafx.com/fxml" hgap="20" vgap="10">
  <children>
    <TextField fx:id="input" GridPane.columnIndex="0" GridPane.rowIndex="0"/>
    <Button text="Converti" GridPane.columnIndex="1" GridPane.rowIndex="1"/>
    <Label fx:id="output" GridPane.columnIndex="2" GridPane.rowIndex="2"/>
  </children>
</GridPane>
```

2.2 Controlli

I **controlli** di base permettono di visualizzare dati e impostazioni e consentono all'utente di effettuare scelte e azioni e di inserire dati. Tra i più utilizzati per la realizzazione di una semplice GUI ci sono pulsanti, *check-box*, *choice-box*, *label* e campi di testo.

La FIGURA 11 visualizza l'aspetto grafico di alcuni controlli JavaFX.

FIGURA 11

2.2.1 *Button*

Il controllo di tipo *button* (classe *Button* del *package javafx.scene.control*; elemento `<Button>` nel file FXML) è visualizzato come un pulsante con un'etichetta grafica o testuale e il suo uso tipico è quello di essere usato dall'utente per generare un evento *MouseEvent.MOUSE_CLICKED* il cui *handler* esegue il codice che implementa l'azione prevista per la pressione del pulsante.

ESEMPIO

Il codice FXML di questo esempio definisce una scena per un'applicazione di conversione tra unità di misura diverse in cui il campo di testo *input* contiene il dato da convertire, la *label output* il risultato della conversione e il pulsante scatena l'evento di tipo *MouseEvent.MOUSE_ CLICKED*, il cui *handler handleConversion* effettua la conversione acquisendo il dato dal campo di testo e visualizzando il risultato nella *label*:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.TextField?>
<?import javafx.scene.layout.Label?>
<?import javafx.scene.layout.Button?>

<Hbox prefHeight="200" prefWidth="400" spacing="50"
     xmlns:fx="http://javafx.com/fxml"
     fx:controller="com.company.controller.HomeController">
  <padding>
      <Insets top="25" right="25" bottom="25" left="25"/>
  </padding>
  <children>
    <TextField fx:id="input" prefWidth="100"/>
    <Button onMouseClicked="#handleConversion" prefWidth="100" text="Converti"/>
    <Label fx:id="output" prefWidth="150" text="${controller.outputValue}"/>
  </children>
</Hbox>
```

Il risultato viene visualizzato nella *label output* per la quale è stato effettuato un *binding* con la proprietà *outputValue* del *controller*.

L'etichetta di testo di un pulsante può essere modificata dinamicamente da codice invocando il metodo *setText*:

```
@FXML Button button;
...
```

```
...
...
button.setText("Stop");
...
```

2.2.2 *Check-box*

Il controllo di tipo *check-box* (classe *CheckBox* del *package javafx.scene.control*; elemento `<CheckBox>` nel file FXML) consente all'utente di effettuare una scelta binaria, può infatti assumere solo i valori «selezionato» o «non selezionato» se lo stato è «determinato», nel caso che lo stato sia «indeterminato» non assume un valore valido.

La tabella che segue elenca i metodi fondamentali della classe *CheckBox*.

Metodo	Descrizione
`isIndeterminate()`	Restituisce **`true`** se lo stato del *check-box* è indeterminato, **`false`** altrimenti.
`isSelected()`	Restituisce **`true`** se il *check-box* è selezionato, **`false`** altrimenti.
`setAllowIndeterminate(boolean value)`	Permette di abilitare/disabilitare la possibilità di stato indeterminato.
`setIndeterminate(boolean value)`	Imposta lo stato del *check-box*: indeterminato o meno.
`setSelected(boolean value)`	Permette di selezionare/deselezionare da codice il *check-box*.

2.2.3 *Choice-box*

Un controllo di tipo *choice-box* (classe *ChoiceBox* del *package javafx.scene.control*; elemento `<ChoiceBox>` nel file FXML) consente all'utente di effettuare una selezione tra un certo numero di opzioni visualizzate in un elenco a scomparsa. Normalmente l'oggetto della selezione non è una stringa di testo che è invece quanto viene visualizzato nell'elenco delle opzioni del controllo; a questo scopo è utile creare una classe che comprenda come attributi sia la stringa da visualizzare nel controllo sia l'oggetto da selezionare.

ESEMPIO La seguente classe permette di rappresentare un multiplo/sottomultiplo di un'unità di misura con il relativo fattore di conversione rispetto all'unità di misura di base; può essere per esempio utilizzata per un controllo *choice-box* che permetta di selezionare il multiplo/sottomultiplo desiderato per effettuare dei calcoli, o una conversione:

```
public class Unità {
  private String nome;
  private Double fattore;

  public Unità(String nome, Double fattore) {
    this.nome = nome;
    this.fattore = fattore;
  }

  public String getNome() {
    return nome;
  }
```

```
    public void setNome(String nome) {
      this.nome = nome;
    }

    public Double getFattore() {
      return fattore;
    }

    public void setFattore(Double fattore) {
      this.fattore = fattore;
    }

    @Override
    public String toString() {
      return nome;
    }
  }
```

L'associazione tra oggetto del linguaggio Java e componente grafico avviene specificando la classe che rappresenta l'oggetto della selezione; con riferimento alla classe *Unità* dell'esempio precedente si può per esempio definire un attributo in questo modo:

```
@FXML ChoiceBox<Unità> unitàMisura;
```

L'inizializzazione di un controllo *choice-box* richiede di specificare l'elenco degli oggetti che costituiscono le opzioni come nella seguente riga di codice:

```
unitàMisura.getItems().addAll(new Unità("Byte", 1.0),
                              new Unità("KiloByte", 1024.0),
                              new Unità("MegaByte", 1048576.0),
                              new Unità("GigaByte", 1073741824.0),
                              new Unità("TeraByte", 1099511627776.0)
                              );
```

OSSERVAZIONE Il metodo *getItems* dell'oggetto di classe *ChoiceBox* restituisce la collezione degli oggetti che costituiscono le opzioni in una forma compatibile con l'interfaccia *List* del *package java.util*: il metodo *addAll* consente di aggiungere un numero variabile di elementi con una sola invocazione.

L'inizializzazione di un controllo *choice-box* impone di fornire al controllo stesso un oggetto di classe derivata dalla classe *StringConverter* del *package javafx.util* che esponga i metodi per trasformare gli oggetti che costituiscono le opzioni in stringhe e viceversa.

ESEMPIO Con riferimento alla classe *Unità* dell'esempio precedente, la relativa classe derivata da *StringConverter* è la seguente:

```
class UnitàStringConverter extends StringConverter<Unità> {
  @Override
  public String toString(Unità unità) {
    return unità.getNome();
  }
```

```
    @Override
    public Unità fromString(String string) {
      return null;
    }
  }
```

Una volta definita, la classe *StringConverter* specifica se deve registrare un'istanza invocando il metodo *setConverter* del controllo di tipo *checkbox*:

```
unitàMisura.setConverter(new UnitàStringConverter());
```

Deve inoltre essere selezionata l'opzione selezionata; a questo scopo il metodo *getSelectionModel* restituisce un oggetto istanza della classe *SingleSelectionModel* del *package javafx.scene.control* che espone i seguenti metodi:

Metodo	Descrizione
`getItemCount()`	Restituisce il numero di opzioni selezionabili dall'utente.
`getSelectedIndex()`	Restituisce l'indice dell'opzione selezionata.
`selectFirst()`	Seleziona la prima opzione.
`selectLast()`	Seleziona l'ultima opzione.
`selectNext()`	Seleziona l'opzione successiva rispetto a quella attualmente selezionata.
`selectPrevious()`	Seleziona l'opzione precedente rispetto a quella attualmente selezionata.
`select(int index)`	Seleziona l'opzione di indice *index*.

L'oggetto associato all'opzione selezionata è restituito dall'invocazione del metodo *getValue*; con riferimento agli esempi precedenti:

```
Unità unità = unitàMisura.getValue();
```

2.2.4 *Label*

Un controllo di tipo *label* (classe *Label* del *package javafx.scene.control*; elemento `<Label>` nel file FXML) consente di visualizzare una stringa di testo in modalità statica definendone il contenuto nell'attributo *text* direttamente nel file FXML di definizione della scena in cui è presente il controllo, oppure in modo dinamico invocando il metodo *setText* che ha come unico argomento la stringa di testo da visualizzare:

```
@FXML Label label;
...
...
...
label.setText("Nessun risultato!");
```

Una *label* viene spesso utilizzata per visualizzare all'utente un risultato o un messaggio: a questo scopo il codice dello *handler* che produce il risultato, o il messaggio, può invocare il metodo *setText*, oppure può essere effettuato un *binding* tra la *label* e una proprietà definita nel codice.

2.2.5 *Text-field*

Un controllo di tipo *text-field* (classe *TextField* del *package javafx.scene.control*; elemento `<TextField>` nel file FXML) consente all'utente di inserire una stringa di testo che viene successivamente processata dal codice, tipicamente dallo *handler* di un evento di click con il mouse su un pulsante.

Nel campo di testo può essere visualizzato un messaggio di invito per la digitazione del contenuto invocando il metodo *setPromptText*, mentre il contenuto del campo è restituito dall'invocazione del metodo *getText*:

```
@FXML TextField input;
...
...
...
input.setPromptText("Inserire lo username");
...
...
...
String username = input.getText();
...
```

Se necessario, il metodo *clear* cancella il contenuto del campo di testo e il metodo *setText* consente di sovrascriverlo con la stringa fornita come argomento.

Come visto in precedenza, un campo di testo è il tipico controllo con il quale viene creato un doppio *binding* con una proprietà della classe *controller* allo scopo di accedere al suo contenuto accedendo al contenuto della proprietà.

OSSERVAZIONE Un controllo di tipo *password-field* (classe *PasswordField* del *package javafx.scene.control*; elemento `<PasswordField>` nel file FXML) è analogo al controllo di tipo *text-field* (si tratta di una classe derivata), ma il testo digitato nel controllo non viene visualizzato.

2.3 Applicazione di esempio

Per esemplificare l'uso dei *container* e dei *controller* JavaFX prendiamo in esame il codice di un'applicazione completa: un convertitore di unità di misura della capacità di memoria il cui codice è organizzato in base al *pattern* MVC (FIGURA 12).

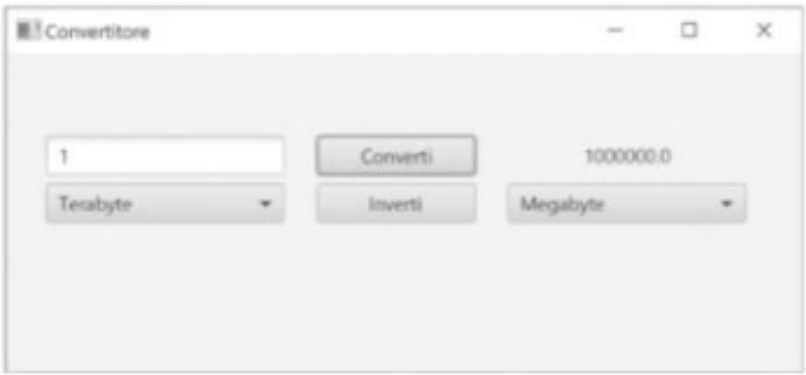


FIGURA 12

2.3.1 *View*

Il seguente file FXML, denominato *home.fxml*, è stato generato dallo strumento *Scene Builder* usato per progettare l'interfaccia grafica dell'applicazione:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.ChoiceBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.VBox?>

<VBox alignment="CENTER" prefHeight="200" prefWidth="500" spacing="25"
    xmlns:fx="http://javafx.com/fxml"
    fx:controller="com.company.controller.HomeController">
  <padding>
      <Insets bottom="25" left="25" right="25" top="25"/>
  </padding>
  <children>
      <GridPane hgap="20" vgap="5">
      <children>
        <TextField fx:id="inputValue" prefWidth="150"
            GridPane.columnIndex="0" GridPane.rowIndex="0"/>
        <ChoiceBox fx:id="inputUnit" prefWidth="150"
            GridPane.columnIndex="0" GridPane.rowIndex="1"/>
        <Button onMouseClicked="#handleConversione" prefWidth="100"
          text="Converti"
          GridPane.columnIndex="1" GridPane.rowIndex="0"/>
        <Button onMouseClicked="#handleInverti" prefWidth="100"
          text="Inverti"
          GridPane.columnIndex="1" GridPane.rowIndex="1"/>
        <Label fx:id="outputValue" alignment="CENTER" prefWidth="150"
          text="${controller.outputValue}" textAlignment="CENTER"
          GridPane.columnIndex="2" GridPane.rowIndex="0"/>
        <ChoiceBox fx:id="outputUnit" prefWidth="150"
            GridPane.columnIndex="2" GridPane.rowIndex="1"/>
      </children>
      </GridPane>
    <Label text="${controller.errorMessage}" textAlignment="CENTER"
        textFill="red"/>
  </children>
</VBox>
```

I seguenti componenti grafici – per ciascuno dei quali è stata definita una larghezza di riferimento – sono disposti in un contenitore di tipo *V-box* il cui primo elemento è a sua volta un contenitore di tipo *grid pane*:

- una *label* anonima per gli eventuali messaggi di errore con *binding* alla proprietà *errorMessage* della classe *HomeController* del *package com.company.controller* nel secondo elemento del contenitore *V-box*;
- un *text-field* (*InputValue*) per il dato da convertire nella cella (0,0) del contenitore *grid pane*;

- una *label* (*outputValue*) per il risultato della conversione con *binding* alla proprietà *outputValue* della classe *HomeController* del *package com.company.controller* nella cella (2,0) del contenitore *grid pane*;
- due *choiche-box* (*inputUnit* e *outputUnit*) rispettivamente per la selezione dell'unità di misura del dato e del risultato della conversione rispettivamente nelle celle (0,1) e (2,1) del contenitore *grid pane*;
- un pulsante anonimo con testo «Converti» che al click del mouse invoca il metodo *handleConversione* della classe *HomeController* del *package com.company.controller* nella cella (1,0) del contenitore *grid pane*;
- un pulsante anonimo con testo «Inverti» che al click del mouse invoca il metodo *handleInverti* della classe *HomeController* del *package com.company.controller* nella cella (1,1) del contenitore *grid pane*.

2.3.2 Model

Il *model* dell'applicazione è costituito dalla classe *Unità* introdotta in un esempio precedente e contenuta nel *package com.company.model*.

2.3.3 Controller

La classe *HomeController* del *package com.company.controller* implementa il *controller* dell'applicazione riferendo gli elementi FXML e definendo le proprietà necessarie per i *binding* previsti:

```
import com.company.model.Unità;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.fxml.FXML;
import javafx.scene.control.ChoiceBox;
import javafx.scene.control.TextField;
import javafx.scene.input.MouseEvent;
import javafx.util.StringConverter;

public class HomeController {

  /* proprietà */
  private DoubleProperty outputValue = new SimpleDoubleProperty(0.00);
  private StringProperty errorMessage = new SimpleStringProperty();

  /* riferimenti FXML */
  @FXML TextField inputValue;
  @FXML ChoiceBox<Unità> inputUnit;
  @FXML ChoiceBox<Unità> outputUnit;

  /* classe StringConverter */
  private class UnitàStringConverter extends StringConverter<Unità> {
    @Override
    public String toString(Unità unità) {
      return unità.getNome();
    }
```

```
    @Override
    public Unità fromString(String string) {
      return null;
    }
  }

  @FXML
  public void initialize() {
    // imposta le opzioni dei choice-box
    inputUnit.getItems().addAll(new Unità("Byte", 1.0),
        new Unità("Kilobyte", 1024.0),
        new Unità("Megabyte", 1048576.0),
        new Unità("Gigabyte", 1073741824.0),
        new Unità("Terabyte", 1099511627776.0)
    );
    outputUnit.getItems().addAll(new Unità("Byte", 1.0),
        new Unità("Kilobyte", 1024.0),
        new Unità("Megabyte", 1048576.0),
        new Unità("Gigabyte", 1073741824.0),
        new Unità("Terabyte", 1099511627776.0)
    );
    // imposta i convertitori dei choice-box
    inputUnit.setConverter(new UnitàStringConverter());
    outputUnit.setConverter(new UnitàStringConverter());
    // seleziona la prima unità di misura per i choice-box
    inputUnit.getSelectionModel().selectFirst();
    outputUnit.getSelectionModel().selectFirst();
  }

  /* event handler che gestisce la conversione */
  public void handleConversione(MouseEvent mouseEvent) {
    Double dato = getDato();
    Unità unitàDato = inputUnit.getValue();
    Unità unitàRisultato = outputUnit.getValue();
    Double risultato = dato /
      (unitàRisultato.getFattore() / unitàDato.getFattore());
    setOutputValue(risultato);
  }

  /* event handler che inverte i campi */
  public void handleInverti(MouseEvent mouseEvent) {
    // scambia unità di misura
    int indiceUnitàDato = inputUnit.getSelectionModel().getSelectedIndex();
    int indiceUnitàRisultato = outputUnit.getSelectionModel().getSelectedIndex();
    inputUnit.getSelectionModel().select(indiceUnitàRisultato);
    outputUnit.getSelectionModel().select(indiceUnitàDato);
    // scambia valori
    Double dato = getDato();
    Double risultato = getOutputValue();
    inputValue.textProperty().setValue(risultato.toString());
    setOutputValue(dato);
  }

  /* metodo di utilità per acquisire il dato di input */
  private Double getDato() {
    Double dato = 0.00;
```

```
    try {
      dato = Double.parseDouble(this.inputValue.textProperty().get());
      setErrorMessage("");
    }
    catch(NumberFormatException eccezione) {
      setErrorMessage("Dato non valido");
    }
    return dato;
  }

  /* getter e setter delle proprietà */
  public double getOutputValue() {
    return outputValue.get();
  }

  public DoubleProperty outputValueProperty() {
    return outputValue;
  }

  public void setOutputValue(double outputValue) {
    this.outputValue.set(outputValue);
  }

  public String getErrorMessage() {
    return errorMessage.get();
  }

  public StringProperty errorMessageProperty() {
    return errorMessage;
  }

  public void setErrorMessage(String errorMessage) {
    this.errorMessage.set(errorMessage);
  }
}
```

Il metodo *initialize* configura i *check-box* di selezione delle unità di misura utilizzando la classe *StringConverter* vista in un esempio precedente definita come classe interna al *controller* stesso.

I due *handler* implementano il comportamento dell'applicazione al verificarsi dell'evento *MouseEvent.MOUSE_CLICKED* rispettivamente sui due pulsanti della GUI: uno per effettuare la conversione (*handleConversione*) e uno per invertire il dato con il risultato e le rispettive unità di misura (*handleInverti*).

Il metodo di utilità *getDato* effettua la conversione tra la stringa digitata dall'utente nel campo di testo e il valore numerico necessario per il calcolo gestendo l'eventuale impossibilità di conversione con l'impostazione di un messaggio di errore visualizzato nella GUI.

2.3.4 Applicazione JavaFX

La classe *Convertitore* del *package com.company* è la classe principale dell'applicazione JavaFX e, come tale, ha il compito di caricare il file FXML e di utilizzarlo per inizializzare la scena dello *stage* e, successivamente, visualizzarla:

```java
import java.io.IOException;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Convertitore extends Application {

  public Convertitore() { }

  @Override
  public void start(Stage stage) throws IOException {
    Parent root = FXMLLoader.load(getClass().getResource("view/home.fxml"));
    stage.setTitle("Convertitore");
    stage.setScene(new Scene(root));
    stage.show();
  }

  public static void main(String[] args) { launch(args); }
}
```

3 Collezioni JavaFX

JavaFX rende disponibili specifici componenti grafici per la visualizzazione di collezioni di oggetti: per esempio *TableView*, per la visualizzazione in forma di tabella, e *ListView*, per la visualizzazione di liste. Analogamente alle proprietà JavaFX, anche in questo caso risulta utile incorporare le collezioni standard del linguaggio Java all'interno di apposite classi *wrapper* JavaFX: questa soluzione permette al componente grafico di interagire direttamente con la collezione, per esempio per ordinarla o modificarla.

Nel seguito del paragrafo saranno introdotti la collezione *ObservableList* e il componente grafico *TableView*.

3.1 Collezione *ObservableList*

Per definire una lista che sia visualizzabile dal componente grafico *TableView*, si ricorre alle classi che implementano l'interfaccia *ObservableList* del *package javafx.collections*, la cui implementazione più usata è la classe concreta generata dall'invocazione del metodo statico *observableArrayList* della classe *FXCollections*:

```java
ObservableList<...> collezione =
FXCollections.observableArrayList();
```

Una collezione di questo tipo si comporta analogamente alla collezione *ArrayList* del *package java.util* e ne espone gli stessi metodi principali. Per esempio, per aggiungere un elemento alla collezione si impiega il metodo *add*:

```java
collezione.add(...);
```

Analogamente a quanto avviene per *ArrayList*, anche *ObservableList* ri-

chiede di specificare in modo parametrico il tipo degli elementi inseriti nella lista.

3.2 Componente grafico *TableView*

Il componente grafico *TableView* di JavaFX è generalmente utilizzato per realizzare una tabella in cui ogni riga mostra informazioni relative a un oggetto di una collezione *ObservableList*. Nel file FXML risulta importante assegnare un *id* a ogni colonna della tabella in modo che sia possibile riferire ciascuna colonna dal codice della classe *controller*, responsabile del collegamento fra la collezione e il componente *TableView*.

ESEMPIO Il seguente codice mostra un esempio di *TableView* con tre colonne:

```
<TableView fx:id="tabellaDipendenti">
  <columns>
    <TableColumn fx:id="colonnaNome" text="Nome"/>
    <TableColumn fx:id="colonnaCognome" text="Cognome"/>
    <TableColumn fx:id="colonnaCodiceFiscale" text="Codice Fiscale"/>
  </columns>
</TableView>
```

La classe *controller* può riferire la tabella e le singole colonne definendo attributi preceduti dall'annotazione `@FXML`. Le classi *TableView* e *TableColumn* richiedono di specificare il tipo degli elementi contenuti nella collezione e, nel caso di *TableColumn*, anche il tipo della proprietà dell'oggetto della collezione che sarà visualizzato all'interno della colonna stessa.

ESEMPIO Nell'ipotesi che la classe *Dipendente* abbia tre proprietà testuali denominate rispettivamente *nome*, *cognome* e *codiceFiscale* l'associazione tra il componente grafico *TableView* e le proprietà della classe *controller* può essere così effettuata:

```
public class ClasseController {
  /* riferimenti FXML */
  @FXML private TableView<Dipendente> tabellaDipendenti;
  @FXML private TableColumn<Dipendente, String> colonnaNome;
  @FXML private TableColumn<Dipendente, String> colonnaCognome;
  @FXML private TableColumn<Dipendente, String> colonnaCodiceFiscale;
  ...
  ...
  ...
}
```

Per impostare la collezione che si desidera visualizzare all'interno della tabella si usa il metodo *setItems* dell'oggetto istanza della classe *TableView*, inoltre per ciascuna colonna della tabella è necessario specificare quale proprietà degli oggetti della lista deve essere visualizzata invocando il metodo *setCellValueFactory* della classe *TableColumn*; con riferimento all'esempio precedente:

```
public class ClasseController {
  /* riferimenti FXML */
  ...
```

3.3 Un'applicazione CRUD di esempio

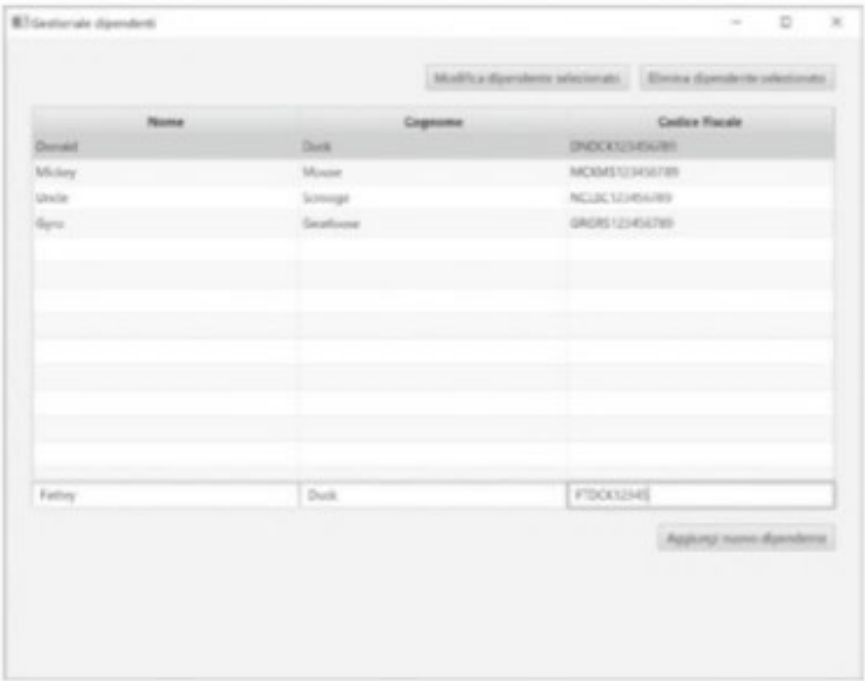
Un caso tipico di utilizzo del componente grafico *TableView* è quello delle applicazioni di tipo CRUD (*Create*, *Read*, *Update* e *Delete*); questa tipologia di applicazioni prevede la possibilità di:

- aggiungere un nuovo oggetto a una collezione digitando il valore delle sue proprietà;
- elencare gli oggetti di una collezione visualizzando il valore delle loro proprietà;
- modificare il valore delle proprietà di un oggetto della collezione;
- eliminare un oggetto della collezione.

Si presenta nel seguito un'applicazione JavaFX di tipo CRUD per la gestione dei dipendenti; in base al *pattern* MVC il codice dell'applicazione è suddiviso in tre distinti *package*: *model*, *controller* e *view*; denominando il *package* principale dell'applicazione *com.company*, i *package* assumeranno rispettivamente i nomi: *com.company.model*, *com.company.controller* e *com.company.view*.

Il modo scelto per realizzare l'applicazione consiste nello scomporla nelle seguenti due *view*:

- finestra con tabella dei dipendenti: *view* che contiene i seguenti elementi:
  - tabella in cui è visualizzato l'elenco dei dipendenti,
  - *form* per i dati necessari all'aggiunta di un nuovo dipendente,
  - pulsanti per l'aggiunta, la rimozione e l'aggiornamento dei dati di un dipendente;
- finestra di modifica dipendente: *view* che contiene il *form* dei dati di un dipendente da modificare.



La *view* di modifica dipendente può essere mostrata in uno *stage* diverso da quello della tabella dei dipendenti creando una finestra *pop-up* che si apre

```
  ...
  ...
  /* lista dei dipendenti */
  private static final ObservableList<Dipendente> dipendenti =
                       FXCollections.observableArrayList();

  public ClasseController() {
    // eventuale inserimento di elementi nella lista
    ...
    ...
    ...
  }

  @FXML
  private void initialize() {
    // impostazione collezione per la tabella di visulizzazione dei dipendenti
    dipendentiTable.setItems(dipendenti);
    colonnaNome.setCellValueFactory(
          new PropertyValueFactory<Dipendente, String>("nome")
    );

    colonnaCognome.setCellValueFactory(
          new PropertyValueFactory<Dipendente, String>("cognome")
    );
    colonnaCodiceFiscale.setCellValueFactory(
                new PropertyValueFactory<Dipendente, String>("codiceFiscale")
    );
  }

  ...
  ...
  ...
}
```

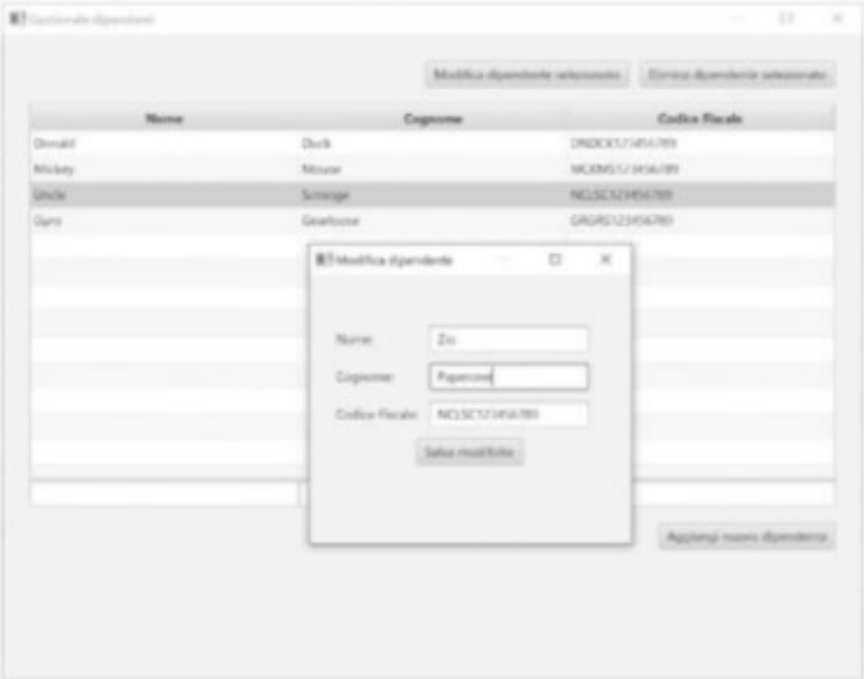
La tabella costruita dal codice precedente assume il seguente aspetto grafico (FIGURA 13).

Nome	Cognome	Codice Fiscale
Donald	Duck	DNDCK123456789
Mickey	Mouse	MCKMS123456789
Uncle	Scrooge	NCLSC123456789
Gyro	Gearloose	GRGRS123456789

FIGURA 13

OSSERVAZIONE Il metodo *setCellValueFactory* richiede come parametro un oggetto istanza della classe di utilità *PropertyValueFactory* del *package javafx.scene.control.cell*. La dichiarazione di un'istanza di questa classe richiede di specificare il tipo degli oggetti della collezione, nell'esempio *Dipendente*, e quello della proprietà associata alla colonna, nell'esempio *String*. Il costruttore richiede come parametro il nome della proprietà degli oggetti della collezione che si intende visualizzare nelle celle della colonna. Questo parametro viene restituito dal metodo *getProperty* dell'oggetto.

quando l'utente esegue il click sul pulsante di modifica dei dati del dipendente selezionato nella tabella.



La classe principale comprende l'istanza di *ObservableList* che contiene i dati relativi ai dipendenti e i metodi necessari a elencare, aggiungere, modificare ed eliminare i dati relativi ai dipendenti nella lista. Il metodo *start* contiene inoltre il codice necessario a impostare i due *stage* necessari all'applicazione e a caricare le relative *scene* FXML.

3.3.1 *Model*

La classe *Dipendente* definisce come proprietà JavaFX gli attributi *nome*, *cognome* e *codiceFiscale* in modo da permettere il *binding* con le colonne del componente grafico *TableView*:

```
package com.company.model;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

/* classe che rappresenta un dipendente */
public class Dipendente {

  /* proprietà */
  private final int id;
  private final StringProperty nome;
  private final StringProperty cognome;
  private final StringProperty codiceFiscale;

  /* costruttori */
  public Dipendente(int id, String nome, String cognome,
                    String codiceFiscale) {
    this.id = id; // identificatore univoco
    this.nome = new SimpleStringProperty(nome);
    this.cognome = new SimpleStringProperty(cognome);
```

```
    this.codiceFiscale = new SimpleStringProperty
                        (codiceFiscale);
  }

  public Dipendente(Dipendente dipendente) {
    this.id = dipendente.getId();
    this.nome = dipendente.nomeProperty();
    this.cognome = dipendente.cognomeProperty();
    this.codiceFiscale = dipendente.codiceFiscaleProperty();
  }

  /* getter e setter */

  public int getId() {
    return id;
  }

  public String getNome() {
    return nome.get();
  }

  public StringProperty nomeProperty() {
    return nome;
  }

  public void setNome(String nome) {
    this.nome.set(nome);
  }

  public String getCognome() {
    return cognome.get();
  }

  public StringProperty cognomeProperty() {
    return cognome;
  }

  public void setCognome(String cognome) {
    this.cognome.set(cognome);
  }

  public String getCodiceFiscale() {
    return codiceFiscale.get();
  }

  public StringProperty codiceFiscaleProperty() {
    return codiceFiscale;
  }

  public void setCodiceFiscale(String codiceFiscale) {
    this.codiceFiscale.set(codiceFiscale);
  }

  @Override
  public boolean equals(Object altro) {
    if (altro == null) return false;
    if (altro == this) return true;
    if (!(altro instanceof Dipendente)) return false;
    Dipendente altroDipendente = (Dipendente) altro;
```

```
    return altroDipendente.getCodiceFiscale()
        .equalsIgnoreCase(this.getCodiceFiscale());
  }
}
```

3.3.2 *View*

La finestra con la tabella dei dipendenti ha come componente grafico fondamentale un *TableView* con vincolo CONSTRAINED_RESIZE_POLICY che visualizza le sole colonne utilizzate; la finestra ha inoltre tre campi di testo, corrispondenti al nome, cognome e codice fiscale di un dipendente, che permettono l'inserimento di un nuovo dipendente. Lo strumento *Scene Builder* genera il seguente codice FXML contenuto nel file *tabellaDipendenti.fxml*:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TableColumn?>
<?import javafx.scene.control.TableView?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.RowConstraints?>
<?import javafx.scene.layout.VBox?>
<?import javafx.geometry.Insets?>

<?import javafx.scene.layout.HBox?>
<VBox prefHeight="600" prefWidth="800" xmlns:fx="http://javafx.com/fxml"
      fx:controller="com.company.controller.TabellaDipendentiController">
  <padding>
    <Insets top="25" right="25" bottom="25" left="25"/>
  </padding>
  <children>
    <VBox alignment="TOP_RIGHT" spacing="15">
      <children>
        <HBox alignment="TOP_RIGHT" spacing="10">
          <children>
            <Button onMouseClicked="#handleModificaDipendente"
                    text="Modifica dipendente selezionato"/>
            <Button onMouseClicked="#handleEliminaDipendente"
                    text="Elimina dipendente selezionato"/>
          </children>
        </HBox>
        <TableView fx:id="tabellaDipendenti" editable="true" maxHeight="350">
          <columnResizePolicy>
            <TableView fx:constant="CONSTRAINED_RESIZE_POLICY"/>
          </columnResizePolicy>
          <columns>
            <TableColumn fx:id="colonnaNome" text="Nome"/>
            <TableColumn fx:id="colonnaCognome" text="Cognome"/>
            <TableColumn fx:id="colonnaCodiceFiscale"
                         text="Codice Fiscale"/>
          </columns>
        </TableView>
```

```
      </children>
    </VBox>
    <VBox alignment="TOP_RIGHT" spacing="15">
      <children>
        <GridPane>
          <columnConstraints>
            <ColumnConstraints hgrow="ALWAYS"/>
            <ColumnConstraints hgrow="ALWAYS"/>
            <ColumnConstraints hgrow="ALWAYS"/>
          </columnConstraints>
          <children>
            <TextField fx:id="campoNome" GridPane.columnIndex="0"/>
            <TextField fx:id="campoCognome" GridPane.columnIndex="1"/>
            <TextField fx:id="campoCodiceFiscale"
                       GridPane.columnIndex="2"/>
          </children>
        </GridPane>
        <Button onMouseClicked="#handleNuovoDipendente"
                text="Aggiungi nuovo dipendente"/>
        <Label text="${controller.messaggioValidazione}"
               textFill="red" textAlignment="RIGHT"/>
      </children>
    </VBox>
  </children>
</VBox>
```

La finestra di modifica di un dipendente è costituita da tre campi di testo, corrispondenti al nome, cognome e codice fiscale di un dipendente, che permettono la modifica dei relativi dati; il pulsante di conferma della modifica salva l'oggetto modificato nella collezione. Lo strumento *Scene Builder* genera il seguente codice FXML contenuto nel file *tabellaDipendenti.fxml*:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.geometry.Insets?>

<VBox prefHeight="250" prefWidth="300" spacing="10" alignment="CENTER"
      xmlns:fx="http://javafx.com/fxml"
      fx:controller="com.company.controller.ModificaDipendenteController">
  <padding>
    <Insets top="25" right="25" bottom="25" left="25"/>
  </padding>
  <children>
    <GridPane vgap="10" hgap="10">
      <children>
        <Label text="Nome:"
               GridPane.columnIndex="0" GridPane.rowIndex="0"/>
        <TextField fx:id="campoNome"
                   GridPane.columnIndex="1" GridPane.rowIndex="0"/>
```

```
        <Label text="Cognome:"
               GridPane.columnIndex="0" GridPane.rowIndex="1"/>
        <TextField fx:id="campoCognome"
                   GridPane.columnIndex="1" GridPane.rowIndex="1"/>
        <Label text="Codice Fiscale:"
               GridPane.columnIndex="0" GridPane.rowIndex="2"/>
        <TextField fx:id="campoCodiceFiscale"
                   GridPane.columnIndex="1" GridPane.rowIndex="2"/>
      </children>
    </GridPane>
    <Button text="Salva modifiche" onMousePressed="#handleSalvaModifiche"/>
    <Label text="${controller.messaggioValidazione}"/>
  </children>
</VBox>
```

3.3.3 *Controller*

La classe *controller* relativa alla finestra con la tabella dei dipendenti inizializza il componente grafico *TableView* e implementa gli *handler* relativi all'evento *MouseEvent.MOUSE_CLICKED* sui pulsanti corrispondenti alle azioni di inserimento di un nuovo dipendente, di modifica del dipendente selezionato e di eliminazione del dipendente selezionato:

```
package com.company.controller;

import com.company.GestionaleDipendenti;
import com.company.model.Dipendente;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.fxml.FXML;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.control.cell.TextFieldTableCell;
import javafx.scene.input.MouseEvent;
import java.util.Random;
import java.util.Date;

public class TabellaDipendentiController {
  /* proprietà */
  private StringProperty messaggioValidazione = new SimpleStringProperty();

  /* riferimenti FXML tabella dipendenti */
  @FXML private TableView<Dipendente> tabellaDipendenti;
  @FXML private TableColumn<Dipendente, String> colonnaNome;
  @FXML private TableColumn<Dipendente, String> colonnaCognome;
  @FXML private TableColumn<Dipendente, String> colonnaCodiceFiscale;

  /* riferimenti FXML form aggiunta dipendente */
  @FXML private TextField campoNome;
  @FXML private TextField campoCognome;
  @FXML private TextField campoCodiceFiscale;

  @FXML
  private void initialize() {
    // impostazioni tabella dei dipendenti
```

```
    tabellaDipendenti.setItems(GestionaleDipendenti.getDipendenti());
    colonnaNome.setCellValueFactory(
      new PropertyValueFactory<Dipendente, String>("nome")
    );
    colonnaCognome.setCellValueFactory(
      new PropertyValueFactory<Dipendente, String>("cognome")
    );
    colonnaCodiceFiscale.setCellValueFactory(
      new PropertyValueFactory<Dipendente, String>("codiceFiscale")
    );
  }

  /* event handler aggiunta nuovo dipendente */
  public void handleNuovoDipendente(MouseEvent mouseEvent) {
    String nome = campoNome.textProperty().get();
    String cognome = campoCognome.textProperty().get();
    String codiceFiscale = campoCodiceFiscale.textProperty().get();

    if (validaDati(nome, cognome, codiceFiscale)) {
      // generazione casuale id nuovo dipendente
      Random random = new Random(new Date().getTime());
      int idNuovoDipendente = random.nextInt();
      Dipendente dipendente = new Dipendente(idNuovoDipendente, nome, cognome,
                                             codiceFiscale);
      GestionaleDipendenti.aggiungiDipendente(dipendente);
      setMessaggioValidazione("Nuovo dipendente aggiunto!");
      campoNome.clear();
      campoCognome.clear();
      campoCodiceFiscale.clear();
    }
  }

  /* event handler modifica dati dipendente selezionato */
  public void handleModificaDipendente(MouseEvent mouseEvent) {
    Dipendente dipendenteSelezionato =
              tabellaDipendenti.getSelectionModel().getSelectedItem();
    if (dipendenteSelezionato == null)
      return;
    // visualizzazione finestra di modifica dati dipendente
    GestionaleDipendenti.mostraPopupModifica(dipendenteSelezionato);
  }

  /* event handler eliminazione dipendente selezionato */
  public void handleEliminaDipendente(MouseEvent mouseEvent) {
    Dipendente dipendenteSelezionato =
              tabellaDipendenti.getSelectionModel().getSelectedItem();
    if (dipendenteSelezionato == null)
      return;
    GestionaleDipendenti.rimuoviDipendente(dipendenteSelezionato);
  }

  /* verifica dati nuovo dipendente */
  private boolean validaDati(String nome, String cognome, String codiceFiscale) {
    boolean valido = true;
    String messaggio = "";

    if (nome == null || nome.isEmpty()) {
        messaggio += "Inserisci nome\n";
```

```
        valido = false;
    }

    if (cognome == null || cognome.isEmpty()) {
        messaggio += "Inserisci cognome\n";
        valido = false;
    }

    if (codiceFiscale == null || codiceFiscale.isEmpty()) {
        messaggio += "Inserisci codice fiscale\n";
        valido = false;
    }

    setMessaggioValidazione(messaggio);
    return valido;
  }

  /* getter e setter */
  public String getMessaggioValidazione() {
    return messaggioValidazione.get();
  }

  public StringProperty messaggioValidazioneProperty() {
    return messaggioValidazione;
  }

  public void setMessaggioValidazione(String messaggio) {
    this.messaggioValidazione.set(messaggio);
  }
}
```

La classe *controller* relativa alla finestra di modifica dei dati di un dipendente implementa lo *handler* relativo all'evento *MouseEvent.MOUSE_CLICKED* sul pulsante di conferma delle modifiche:

```
package com.company.controller;

import com.company.GestionaleDipendenti;
import com.company.model.Dipendente;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.fxml.FXML;
import javafx.scene.control.TextField;
import javafx.scene.input.MouseEvent;

public class ModificaDipendenteController {
  /* proprietà */
  private Dipendente dipendente;
  private StringProperty messaggioValidazione = new SimpleStringProperty();

  /* riferimenti FXML */
  @FXML private TextField campoNome;
  @FXML private TextField campoCognome;
  @FXML private TextField campoCodiceFiscale;

  @FXML
  public void start() {
  }
```

```java
  /* event handler conferma modifiche */
  public void handleSalvaModifiche(MouseEvent mouseEvent) {
    String nome = campoNome.textProperty().get();
    String cognome = campoCognome.textProperty().get();
    String codiceFiscale = campoCodiceFiscale.textProperty().get();

    if (validaDati(nome, cognome, codiceFiscale)) {
        dipendente.setNome(nome);
        dipendente.setCognome(cognome);
        dipendente.setCodiceFiscale(codiceFiscale);
        GestionaleDipendenti.modificaDipendente(dipendente);
        setMessaggioValidazione("Modifiche salvate con successo");
    }
  }

  /* verifica dati dipendente */
  private boolean validaDati(String nome, String cognome, String codiceFiscale) {
    boolean valido = true;
    String messaggio = "";

    if (nome == null || nome.isEmpty()) {
        messaggio += "Inserisci nome\n";
        valido = false;
    }

    if (cognome == null || cognome.isEmpty()) {
        messaggio += "Inserisci cognome\n";
        valido = false;
    }

    if (codiceFiscale == null || codiceFiscale.isEmpty()) {
        messaggio += "Inserisci codice fiscale\n";
        valido = false;
    }

    setMessaggioValidazione(messaggio);
    return valido;
  }

  /* getter e setter */
  public void setDipendente(Dipendente dipendente) {
    this.dipendente = dipendente;
    // visualizzazione dei dati del dipendente
    campoNome.textProperty().setValue(dipendente.getNome());
    campoCognome.textProperty().set(dipendente.getCognome());
    campoCodiceFiscale.textProperty().set(dipendente.getCodiceFiscale());
  }

  public String getMessaggioValidazione() {
    return messaggioValidazione.get();
  }

  public StringProperty messaggioValidazioneProperty() {
    return messaggioValidazione;
  }

  public void setMessaggioValidazione(String messaggio) {
    this.messaggioValidazione.set(messaggio);
  }
}
```

3.3.4 Applicazione

La classe *GestionaleDipendenti* realizza l'applicazione JavaFX di tipo CRUD relativa alla finestra con la tabella dei dipendenti, inizializza il componente grafico *TableView* e implementa gli *handler* relativi all'evento *MouseEvent.MOUSE_CLICKED* sui pulsanti corrispondenti alle azioni di inserimento di un nuovo dipendente, di modifica del dipendente selezionato e di eliminazione del dipendente selezionato. Si noti come la predisposizione della finestra di *pop-up* per la modifica dei dati di un dipendente richiede una nuova istanza di *FXMLLoader* e come l'invocazione del metodo *getController* consente di mantenere un riferimento al *controller* della finestra *pop-up*:

```
package com.company;

import com.company.controller.ModificaDipendenteController;
import com.company.model.Dipendente;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Modality;
import javafx.stage.Stage;
import java.util.Random;
import java.util.Date;
import java.io.IOException;

public class GestionaleDipendenti extends Application {
  /* lista dei dipendenti */
  private static final ObservableList<Dipendente> dipendenti =
                      FXCollections.observableArrayList();
  /* pop-up stage e controller per modifica dipendente */
  private static Stage popupStage;
  private static ModificaDipendenteController popupController;

  /* costruttore: inserisce dati per test nella collezione */
  public GestionaleDipendenti() {
    // generatore casuale id nuovo dipendente
    Random random = new Random(new Date().getTime());

    dipendenti.add(new Dipendente(random.nextInt(),
                                  "Donald", "Duck", "DNDCK123456789"));
    dipendenti.add(new Dipendente(random.nextInt(),
                                  "Mickey", "Mouse", "MCKMS123456789"));
    dipendenti.add(new Dipendente(random.nextInt(),
                                  "Uncle", "Scrooge", "NCLSC123456789"));
    dipendenti.add(new Dipendente(random.nextInt(),
                                  "Gyro", "Gearloose", "GRGRS123456789"));
  }

  @Override
  public void start(Stage primaryStage) throws IOException {
    // predisposizione stage principale
    Parent root =
```

```
    FXMLLoader.load(getClass().getResource("view/tabellaDipendenti.fxml"));
    primaryStage.setTitle("Gestionale dipendenti");
    primaryStage.setScene(new Scene(root));
    // predisposizione stage pop-up
    popupStage = new Stage();
    FXMLLoader fxmlLoader = new
    FXMLLoader(getClass().getResource("view/modificaDipendente.fxml"));
    Parent popupRoot = fxmlLoader.load();
    // registrazione riferimento controller pop-up
    popupController = fxmlLoader.getController();
    popupStage.setScene(new Scene(popupRoot));
    popupStage.setTitle("Modifica dipendente");
    popupStage.initModality(Modality.APPLICATION_MODAL);
    popupStage.initOwner(primaryStage);
    // visualizzazione stage principale
    primaryStage.show();
  }

  public static void mostraPopupModifica(Dipendente dipendente) {
    popupController.setDipendente(dipendente);
    popupStage.showAndWait();
  }

  public static ObservableList<Dipendente> getDipendenti() {
    return dipendenti;
  }

  public static void aggiungiDipendente(Dipendente dipendente) {
    dipendenti.add(dipendente);
  }

  public static void rimuoviDipendente(Dipendente dipendente) {
    dipendenti.remove(dipendente);
  }

  public static void modificaDipendente(Dipendente dipendenteAggiornato) {
    for (Dipendente dipendente : dipendenti) {
      if (dipendente.getId() == dipendenteAggiornato.getId()) {
          dipendente = dipendenteAggiornato;
      }
    }
  }

  public static void main(String[] args) {
    launch(args);
  }
}
```

I CONCETTI CHIAVE

INTERFACCIA UTENTE GRAFICA O GUI. La maggior parte delle applicazioni moderne dispone di un'interfaccia grafica o GUI (*Graphical User Interface*) grazie alla quale è possibile mostrare informazioni all'utente utilizzando oggetti grafici interattivi contenuti in una o più «finestre»; inoltre, una GUI permette modalità di input diverse dalla sola tastiera, per esempio: mouse, touchscreen, joystick, ecc.

JAVAFX. JavaFX è un insieme di librerie e componenti standard per la realizzazione di interfacce grafiche per applicazioni Java. È incluso all'interno della distribuzione del linguaggio Java e, di conseguenza, un'applicazione JavaFX, a differenza di altri *framework* per la realizzazione di GUI, non necessita di librerie o componenti aggiuntivi per la propria esecuzione su un sistema su cui è installato il supporto Java.

***PATTERN* MVC.** Analogamente ad altri *framework* per la realizzazione di GUI, JavaFX rende disponibili strumenti che facilitano l'implementazione del *pattern* architetturale denominato MVC (*Model-View-Controller*). Il *pattern* MVC prevede la scomposizione di un'applicazione grafica in tre tipi di componenti:

- *Model*: logica interna, indipendente dall'interfaccia grafica, spesso riferita come *business logic*;
- *View*: interfaccia grafica dell'applicazione;
- *Controller*: gestione dell'interazione dell'utente con l'interfaccia grafica.

***STAGE* E *SCENE*.** JavaFX adotta l'analogia teatrale per l'attribuzione dei nomi ai propri componenti principali. La finestra che contiene l'applicazione è definita *stage* (palco) e al suo interno sono rappresentate una o più *scene* (scene). Come le scene di uno spettacolo rappresentano momenti e ambientazioni diverse della storia, in JavaFX ogni *scene* rappresenta una diversa sezione, ovvero una diversa *view*, dell'applicazione. Ogni *scene* è caratterizzata da un *node-graph*, ovvero una struttura gerarchica contenente i vari elementi grafici, denominati *node* (nodi) che costituiscono la GUI della scena.

FXML. Sebbene sia possibile costruire interfacce grafiche JavaFX utilizzando solo codice Java, è consigliabile utilizzare invece FXML, un linguaggio di *markup* basato su XML che è parte integrante del *framework* JavaFX. I file che contengono codice FXML non sono normalmente editati dallo sviluppatore, ma generati mediante *Scene Builder*, un editor grafico che permette di progettare l'interfaccia grafica di un'applicazione JavaFX.

INIZIALIZZAZIONE DI UN'APPLICAZIONE JAVAFX. Il codice Java di un'applicazione deve integrare il codice FXML che ne definisce l'interfaccia grafica. A questo scopo, per realizzare un'applicazione grafica con JavaFX, la sua classe principale dovrà estendere la classe *Application* del *package javafx.application* e sovrascriverne il metodo *start* il cui unico parametro è il riferimento allo *stage* (finestra) dell'applicazione stessa. Nel codice del metodo *start* è possibile caricare un file FXML e creare una nuova *scene* (scena) a partire dal contenuto del file FXML caricato invocando il metodo statico *load* della classe *FXMLLoader* del *package javafx.fxml*. Infine si può mostrare la scena sullo *stage*, che comporta il *rendering* dei componenti grafici definiti nel file FXML all'interno della finestra dell'applicazione invocando il metodo *show* dello *stage* principale.

CLASSE *STAGE*. La classe *Stage* del *package javafx.stage* consente di definire e manipolare la finestra di visualizzazione della GUI di un'applicazione JavaFX.

CLASSE *CONTROLLER*. Per assicurare il funzionamento di un'applicazione JavaFX con una GUI che implementi il *pattern* MVC è necessario associare a una scena, costruita a partire dal contenuto di un file FXML, una classe *controller* che gestisca l'interazione fra l'utente e l'interfaccia grafica. Per indicare la classe *controller* nel file FXML si ricorre all'attributo *fx:controller* dell'elemento radice: il valore da assegnare all'attributo è il nome della classe Java che implementa il *controller*.

EVENTI. Il concetto di interfaccia grafica è fortemente legato a quello di evento: gli eventi sono utilizzati per notificare l'applicazione quando l'utente esegue determinate azioni utilizzando i dispositivi di input, per esempio la pressione di un tasto della tastiera, il movimento o il click del mouse, o il tocco su un touchscreen. L'applicazione dotata di GUI può registrare degli *event handler*, ovvero metodi il cui codice deve essere eseguito in corrispondenza del verificarsi di uno specifico evento, come il click del mouse su un pulsante della GUI.

EVENTI JAVAFX. Ogni evento JavaFX è di un particolare tipo e i diversi tipi di evento sono organizzati in una gerarchia di classi. Questa organizzazione gerarchica risulta utile quando si registrano gli *event handler*: oltre a poter registrare un *handler* per uno specifico tipo di evento si può registrare un *handler* più generico per l'intera famiglia di eventi e poi distinguere lo specifico evento nel codice dello *handler*.

PROGRAMMAZIONE A EVENTI. La programmazione a eventi avviene in maniera *asincrona*: il codice dell'applicazione non è eseguito in maniera sequenziale ma è distribuito nei metodi che implementano gli *handler*. Ogni *handler* deve essere registrato specificando un elemento *target* e un tipo di evento: quando un evento del tipo specificato occorre sull'elemento *target* specificato, il

relativo metodo *handler* viene invocato da JavaFX. Una tecnica molto utilizzata è quella di registrare gli *event handler* direttamente all'interno del file FXML di definizione della *scene*. Per farlo si utilizzano degli specifici attributi dei nodi il cui valore è il nome del metodo del *controller* che si desidera registrare come *event handler* preceduto dal simbolo «#».

BINDING. Oltre a gestire i riferimenti a elementi FXML, JavaFX permette di definire dei *binding* (legami) fra le proprietà della classe *controller* e le proprietà degli elementi grafici di una *scene*. Per esempio, effettuando un *binding* fra una proprietà del *controller* e il valore di un campo di testo della *scene*, il valore inserito dall'utente all'interno del campo sarà automaticamente sincronizzato con la relativa proprietà della classe *controller*.

***BINDING* BIDIREZIONALE.** È possibile definire un *binding* bidirezionale, ovvero che consente di modificare una proprietà sia dal codice del *controller* sia mediante l'interazione dell'utente, con un componente grafico della GUI definito in una *scene* FXML. Questo comportamento risulta particolarmente utile nel caso di un campo di testo: con un *binding* bidirezionale è infatti possibile fare in modo che una proprietà del *controller* venga aggiornata quando l'utente cambia il valore del campo. A differenza dei *binding* unidirezionali, un *binding* bidirezionale non può essere definito all'interno del file FXML ma solo all'interno del *controller*.

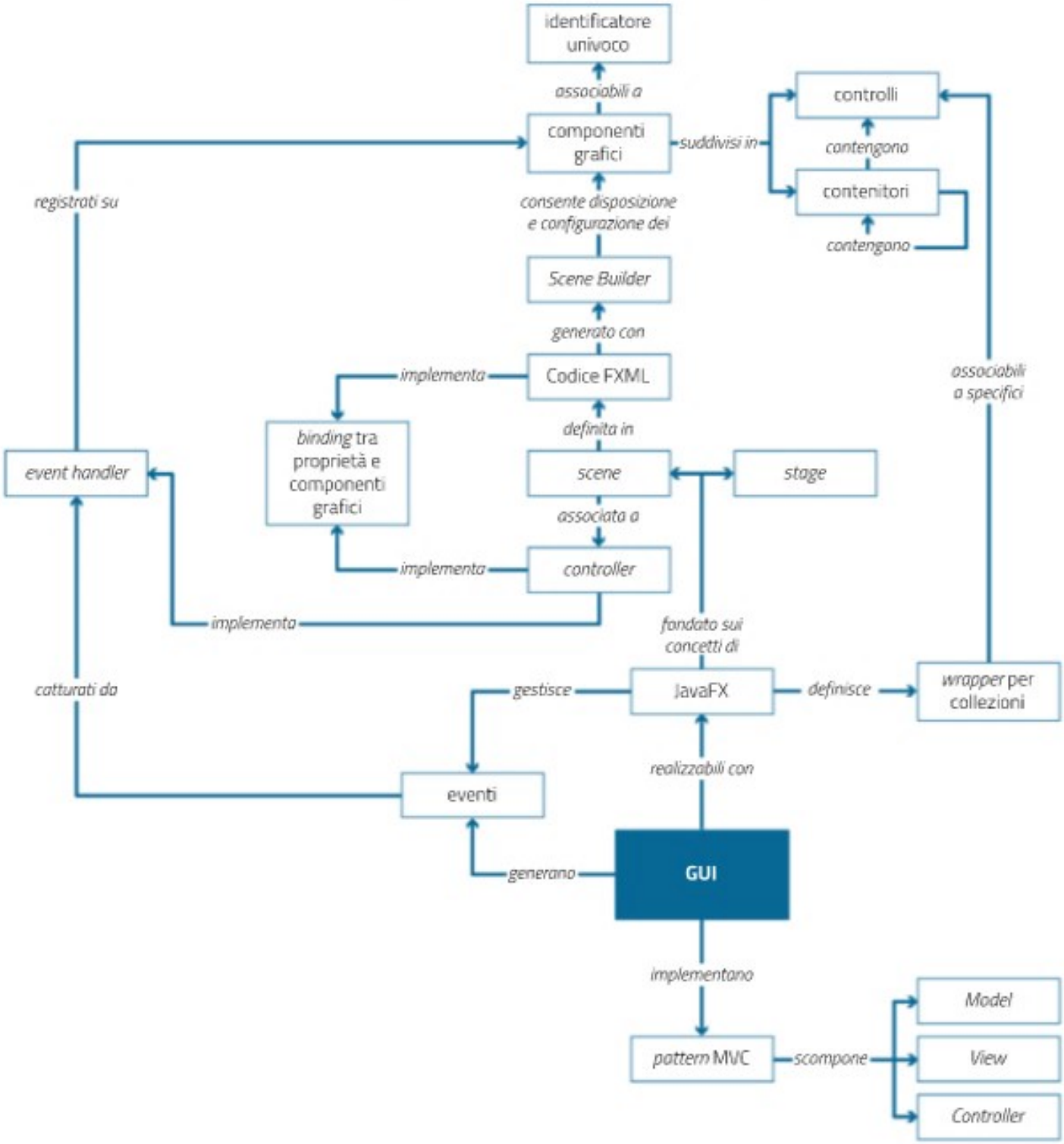
***SCENE BUILDER* E COMPONENTI GRAFICI**. Ogni singola scena di un'applicazione JavaFX è definita da un file FXML; il file viene normalmente generato utilizzando uno strumento integrato nei principali IDE per il linguaggio di programmazione Java, *Scene Builder*, che permette di comporre visualmente il *layout* grafico di una scena inserendovi con l'uso del mouse i componenti grafici resi disponibili da JavaFX e presenti sotto forma di icone nella libreria dei componenti. Una volta inseriti nel *layout* della scena i componenti possono essere singolarmente selezionati e configurati impostandone le relative proprietà che usualmente corrispondono agli attributi dell'elemento corrispondente nel file FXML generato.

CONTENITORI. Un componente di tipo *container* comprende al proprio interno altri componenti, compresi altri *container*: questo permette una strutturazione gerarchica della disposizione dei componenti dell'interfaccia grafica che si riflette nella struttura ad albero del codice FXML generato.

CONTROLLI. I controlli di base permettono di visualizzare dati e impostazioni e consentono all'utente di effettuare scelte e azioni e di inserire dati. Tra i più utilizzati per la realizzazione di una semplice GUI ci sono pulsanti, *check-box*, *choice-box*, *label* e campi di testo.

COLLEZIONI JAVAFX. JavaFX rende disponibili specifici componenti grafici per la visualizzazione di collezioni di oggetti: per esempio *TableView*, per la visualizzazione in forma di tabella, e *ListView*, per la visualizzazione di liste. Analogamente alle proprietà JavaFX, anche in questo caso risulta utile incorporare le collezioni standard del linguaggio Java all'interno di apposite classi *wrapper* JavaFX: questa soluzione permette al componente grafico di interagire direttamente con la collezione, per esempio per ordinarla o modificarla.

RIPASSA CON LA MAPPA



SPIEGA IN UNA PAGINA

1 Descrivere le caratteristiche, le finalità, la strutturazione e gli strumenti del *framework* JavaFX.

2 Descrivere le caratteristiche e l'uso tipico dei principali componenti grafici (contenitori e controlli) del *framework* JavaFX.

QUESITI A RISPOSTA APERTA

3 Qual è lo scopo del linguaggio FXML?

4 Descrivere il *pattern* MVC definendo il ruolo dei componenti *Model*, *View* e *Controller*.

5 Che cos'è la programmazione *event driven*?

TEST A RISPOSTA MULTIPLA

6 JavaFX è:

A la nuova versione del linguaggio di programmazione Java.
B il *framework* standard per la realizzazione di GUI in linguaggio Java.
C la nuova libreria per le collezioni del linguaggio Java.
D un *framework* di terza parte per la realizzazione di GUI in linguaggio Java.

7 Quali sono i componenti di un'applicazione previsti dal *pattern* MVC?

A *model*, *view* e *controller*.
B *stage*, *scene* e *controller*.
C *application*, *controller* e GUI.
D *container*, *control* e *model*.

8 In JavaFX lo *SceneGraph* è:

A il file FXML che definisce una *scene*.
B la collezione ad albero dei dati gestiti da un'applicazione dotata di GUI.
C la collezione di tutte le *scene* che possono essere visualizzate in uno *stage*.
D la struttura gerarchica degli elementi grafici di una *scene*.

9 Quali dei seguenti sono possibili nodi di un *SceneGraph*?

A forme geometriche 2D.
B forme geometriche 3D.
C elementi UI.
D *layout*.

10 Il codice FXML è di solito:

A editato manualmente dallo sviluppatore di una GUI.
B generato con uno strumento di editing grafico della GUI.
C generato dal codice Java dell'applicazione.
D distribuito da appositi *repository* accessibili *online*.

11 Quali delle seguenti azioni sono tipicamente eseguite dal metodo *start* di un'applicazione JavaFX?

A Caricamento dei file FXML di definizione delle *scene*.
B Inizializzazione e configurazione dei componenti grafici di una scena.
C Visualizzazione della prima scena di uno *stage*.
D Registrazione degli *handler* degli eventi.

12 Per quale scopo si può attribuire un identificativo univoco agli elementi della GUI definiti in un file FXML?

A Per riferirli come proprietà della classe *controller* utilizzando l'annotazione **@FXML**.
B Per garantirne l'organizzazione gerarchica.
C Per permettere al caricatore di file FXML di visualizzarli nella *scene*.
D Non è possibile attribuire un identificativo univoco agli elementi di un file FXML.

13 Che cosa non è un *event handler*?

A Un metodo della classe *controller* invocato per associare un evento a un componente grafico della GUI.
B Un componente grafico della GUI con il quale l'utente può interagire.
C Un metodo della classe *controller* invocato automaticamente al verificarsi di un'azione da parte di un utente su un componente grafico della GUI.
D Un metodo della classe *controller* il cui codice

inizializza e configura un componente grafico della GUI.

14 **Qual è l'ordine corretto delle fasi di *dispatch* di un evento in JavaFX?**

A *target selection, route construction, capturing, bubbling.*

B *bubbling, capturing, route construction, target selection.*

C *target selection, capturing, bubbling, route construction.*

D *bubbling, target selection, capturing, route construction.*

15 **Che cos'è un *convenience method* JavaFX?**

A Un *event handler* di convenienza definito dallo sviluppatore.

B Un *event handler* predefinito dal *framework* JavaFX.

C Un attributo di un elemento del file FXML che consente di specificare un *handler* relativo a un evento specifico.

D Un metodo predefinito per la registrazione di uno specifico *event handler*.

16 **I *binding* in JavaFX possono essere usati per:**

A il collegamento tra una proprietà della classe *controller* e una proprietà di un componente grafico della GUI in cui l'utente può inserire o selezionare un valore.

B il collegamento tra una proprietà della classe *controller* e un componente grafico della GUI.

C il collegamento tra un *event handler* della classe *controller* e uno specifico evento registrato su un componente grafico della GUI.

D JavaFX non prevede i *binding*.

17 **In JavaFX il *binding* bidirezionale:**

A è configurabile direttamente nel file FXML.

B è configurabile solo nel codice del *controller*.

C deve essere configurato sia nel file FXML che nel codice del *controller*.

D In JavaFX non è possibile configurare il *binding* bidirezionale.

18 **Quali dei seguenti sono componenti grafici JavaFX di base per lo strumento *Scene Builder*?**

A *Container.*

B *Control.*

C *Menu.*

D *Shape.*

19 **Un componente grafico JavaFX di tipo *container*:**

A può contenere solo componenti grafici di tipo *control.*

B può contenere solo componenti grafici di tipo *container.*

C può contenere componenti grafici sia di tipo *control* sia *container.*

D Un componente grafico di tipo *container* non può contenere altri componenti grafici.

20 **Per consentire all'utente di un'applicazione di inserire dati in formato alfanumerico quale componente grafico JavaFX è indicato?**

A Un *text-field.*

B Una *label.*

C Un *check-box.*

D Un *choice-box.*

PROBLEMI

21 **Realizzare un'applicazione JavaFX dotata di GUI che consenta di convertire un importo in euro in dollari e viceversa; il fattore di conversione deve essere modificabile dall'utente e salvato in un file da cui viene letto automaticamente all'avvio dell'applicazione stessa.**

22 **Utilizzando eventualmente gli algoritmi illustrati nel primo volume del corso, implementare una classe Java che consenta di calcolare il giorno della settimana corrispondente a una data e il numero di giorni trascorsi tra due date. Realizzare un'applicazione JavaFX che consenta all'utente di usufruire delle funzionalità della classe mediante una GUI.**

CASI PRATICI

23 **Realizzare un'applicazione JavaFX dotata di GUI che permetta di effettuare vari tipi di conversione tra unità di misura del sistema anglosassone e unità di misura del sistema metrico decimale. La finestra dell'applicazione deve consentire di selezionare il tipo di conversione e presentare un unico pulsante per effettuarla.**

24 La seguente classe Java implementa una semplice calcolatrice (il metodo *main* ne esemplifica l'uso):

```
public class Calcolatrice {
   private double operando; // ultimo operando inserito
   private double memoria; // accumulatore risultato
   private char operazione; // tipo di operatore aritmetico (+,-,*,/)
   private boolean errore; // errore di calcolo

   public Calcolatrice() {
     memoria=0.0;
     operando=0.0;
     operazione='#'; // nessun operatore impostato
     errore=false;
   }

   // imposta operatore algebrico
   public void setOperazione(char operazione) {
     if (!errore)
       if ((operazione=='+') || (operazione=='-') ||
           (operazione=='*') || (operazione=='/')) {
         this.operazione=operazione;
         setMemoria(getOperando());
       }
   }

   // restituisce ultima operazione aritmetica impostata
   private char getOperazione() {
     return operazione;
   }

   // imposta operando
   public void setOperando(double valore) {
     if (!errore)
       operando=valore;
   }

   // restituisce operando
   public double getOperando() {
     return operando;
   }

   // imposta accumulatore
   private void setMemoria(double valore) {
     memoria=valore;
   }

   // restituisce valore accumulatore
   private double getMemoria() {
     return memoria;
   }

   // cambio segno operando corrente
   public void cambiaSegno() {
```

```
    setOperando(-1.0*getOperando());
  }

  // reset calcolatrice
  public void cancella() {
    setMemoria(0.0);
    operazione='#';
    errore=false;
  }

  // cancella ultimo operando
  public void cancellaUltimoOperando() {
    setOperando(0.0);
    errore=false;
  }

  // addizione
  public void addiziona() {
    if (!errore) {
      setOperando(getMemoria()+operando);
      setMemoria(getOperando());
    }
  }

  // sottrazione
  public void sottrai() {
    if (!errore) {
      setOperando(getMemoria()-operando);
      setMemoria(getOperando());
    }
  }

  // moltiplicazione
  public void moltiplica() {
    if (!errore) {
      setOperando(getMemoria()*operando);
      setMemoria(getOperando());
    }
  }

  // divisione
  public void dividi() {
    if (!errore) {
      setOperando(getMemoria()/operando);
      setMemoria(getOperando());
    }
  }

  // richiesta risultato
  public void uguale() {
    if (!errore) {
      if (operazione=='+') addiziona();
      else if (operazione=='-') sottrai();
      else if (operazione=='*') moltiplica();
```

```
        else if (operazione=='/') dividi();
      }
    }

    // esporta risultato
    public String risultato() {
      if (operando == Double.POSITIVE_INFINITY ||
          operando == Double.NEGATIVE_INFINITY ||
          operando == Double.NaN) {
        errore = true;
      return "ERR";
      }
    else
        return Double.toString(operando);
    }

    public static void main(String[] args) {
      Calcolatrice c = new Calcolatrice();
      c.setOperando(10.0);
      c.setOperazione('+');
      c.setOperando(20.0);
      c.uguale();
      System.out.println(c.risultato());
      c.setOperazione('/');
      c.setOperando(0.0);
      c.uguale();
      System.out.println(c.risultato());
      c.cancella();
      c.setOperando(20.0);
      c.setOperazione('-');
      c.setOperando(10.0);
      c.uguale();
      System.out.println(c.risultato());
      c.setOperando(10.0);
      c.setOperazione('*');
      c.setOperando(1.0);
      System.out.println(c.risultato());
      c.uguale();
    }
}
```

Realizzare un'applicazione JavaFX che implementi la GUI di una calcolatrice con un visore per la visualizzazione dei dati e dei risultati, i tasti delle cifre numeriche, il tasto del separatore decimale, il tasto per il cambiamento di segno, i tasti per le 4 operazioni aritmetiche, il tasto «=» per il risultato e i tasti «C» per la cancellazione generale e «CE» per la cancellazione del solo ultimo operando inserito.

25 **Realizzare un'applicazione JavaFX che implementi una GUI di tipo CRUD per il problema 19 del capitolo A6.**

26 **Realizzare un'applicazione JavaFX che implementi una GUI di tipo CRUD per il caso pratico 21 del capitolo A6.**

27 Realizzare un'applicazione JavaFX che implementi una GUI di tipo CRUD per il caso pratico 24 del capitolo A6.

PROGETTI

28 Realizzare un'applicazione JavaFX che implementi una GUI di tipo CRUD per il progetto 26 del capitolo A6.

29 Realizzare un'applicazione JavaFX che implementi una GUI di tipo CRUD per il progetto 27 del capitolo A6.

availability
disponibilità

features
caratteristiche

performance
prestazione

JavaFX Overview

This chapter provides an overview of the types of applications you can build using JavaFX APIs, where to download the JavaFX libraries, and a high-level information about the key JavaFX features being delivered.

JavaFX is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms.

[...].

JavaFX Applications

Since the JavaFX library is written as a Java API, JavaFX application code can reference APIs from any Java library. For example, JavaFX applications can use Java API libraries to access native system capabilities and connect to server-based middleware applications.

The look and feel of JavaFX applications can be customized. Cascading Style Sheets (CSS) separate appearance and style from implementation so that developers can concentrate on coding. Graphic designers can easily customize the appearance and style of the application through the CSS. If you have a web design background, or if you would like to separate the user interface (UI) and the back-end logic, then you can develop the presentation aspects of the UI in the FXML scripting language and use Java code for the application logic. If you prefer to design UIs without writing code, then use JavaFX *Scene Builder*. As you design the UI, *Scene Builder* creates FXML markup that can be ported to an Integrated Development Environment (IDE) so that developers can add the business logic.

Availability

The JavaFX APIs are available as a fully integrated feature of the Java SE Runtime Environment (JRE) and the Java Development Kit (JDK). Because the JDK is available for all major desktop platforms (Windows, Mac OS X, and Linux), JavaFX applications compiled to JDK 7 and later also run on all the major desktop platforms.

The cross-platform compatibility enables a consistent runtime experience for JavaFX applications developers and users. Oracle ensures synchronized releases and updates on all platforms and offers an extensive support program for companies that run mission-critical applications.

[...].

Key Features

The following features are included in JavaFX 8 and later releases. Items that were introduced in JavaFX 8 release are indicated accordingly:

- **Java APIs**. JavaFX is a Java library that consists of classes and interfaces that are written in Java code. The APIs are designed to be a friendly alternative to Java Virtual Machine (Java VM) languages, such as *JRuby* and *Scala*.
- **FXML and Scene Builder**. FXML is an XML-based declarative markup language for constructing a JavaFX application user interface. A designer can code in FXML or use JavaFX *Scene Builder* to interactively design the graphical user interface (GUI). Scene Builder generates FXML markup that can be ported to an IDE where a developer can add the business logic.
- **WebView**. A web component that uses *WebKitHTML* technology to make it possible to embed web pages within a JavaFX application. JavaScript running in *WebView* can call Java APIs, and Java APIs can call JavaScript running in *WebView*. Support for additional HTML5 features, including Web Sockets, Web Workers, and Web Fonts, and printing capabilities have been added in JavaFX 8. [...].

- **Swing interoperability**. Existing Swing applications can be updated with JavaFX features, such as rich graphics media playback and embedded Web content. The *SwingNode* class, which enables you to embed Swing content into JavaFX applications, has been added in JavaFX 8. […].
- **Built-in UI controls and CSS**. JavaFX provides all the major UI controls that are required to develop a full-featured application. Components can be skinned with standard web technologies such as CSS. The *DatePicker* and *TreeTableView* UI controls are now available with the JavaFX 8 release. […]. Also, the CSS *Styleable* classes have become public API, allowing objects to be styled by CSS.
- **Modena theme**. The *Modena* theme replaces the *Caspian* theme as the default for JavaFX 8 applications. The Caspian theme is still available for your use by adding the `setUserAgentStylesheet(STYLESHEET_CASPIAN)` line in your Application `start()` method. For more information, see the Modena blog at *fxexperience.com*

What Can I Build with JavaFX?

- **3D Graphics Features**. The new API classes for Shape3D (*Box*, *Cylinder*, *MeshView*, and *Sphere* subclasses), *SubScene*, *Material*, *PickResult*, *LightBase* (*AmbientLight* and *PointLight* subclasses), and *SceneAntialiasing* have been added to the 3D Graphics library in JavaFX 8. The Camera API class has also been updated in this release. […].
- **Canvas API**. The Canvas API enables drawing directly within an area of the JavaFX scene that consists of one graphical element (node).
- **Printing API**. The *javafx.print* package has been added in Java SE 8 release and provides the public classes for the JavaFX Printing API.
- **Rich Text Support**. JavaFX 8 brings enhanced text support to JavaFX, including bi-directional text and complex text scripts, such as Thai and Hindu in controls, and multi-line, multi-style text in text nodes.
- **Multitouch Support**. JavaFX provides support for multitouch operations, based on the capabilities of the underlying platform.
- **Hi-DPI support**. JavaFX 8 now supports Hi-DPI displays.
- **Hardware-accelerated graphics pipeline**. JavaFX graphics are based on the graphics rendering pipeline (*Prism*). JavaFX offers smooth graphics that render quickly through *Prism* when it is used with a supported graphics card or graphics processing unit (GPU). If a system does not feature one of the recommended GPUs supported by JavaFX, then *Prism* defaults to the software rendering stack.
- **High-performance media engine**. The media pipeline supports the playback of web multimedia content. It provides a stable, low-latency media framework that is based on the *GStreamer* multimedia framework.
- **Self-contained application deployment model**. Self-contained application packages have all of the application resources and a private copy of the Java and JavaFX runtimes. They are distributed as native installable packages and provide the same installation and launch experience as native applications for that operating system.

[J. Potts, N. Hildebrandt, J. Gordon, C. Castillo, *JavaFX – Getting Started with JavaFX*, Oracle, 2014]

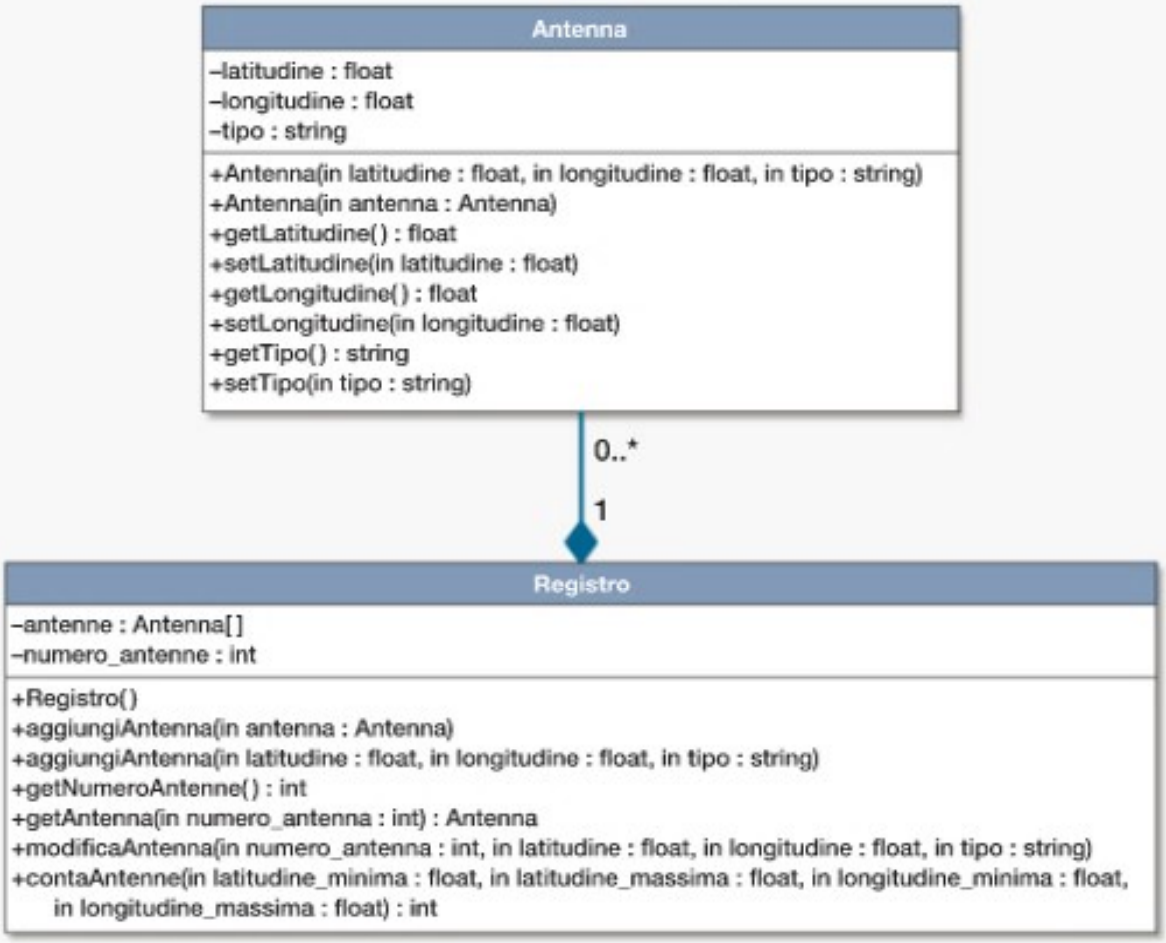
QUESTIONS

a What can be achieved by using CSS with JavaFX applications?
b With JavaFX it is possible to design the UI without coding. How is this possible?
c What is the scope of a *WebView* component?
d Is JavaFX optimized for modern GPUs?

A9

Thread in linguaggio Java

Un operatore di telefonia mobile registra il tipo e le coordinate geografiche della posizione delle antenne della propria rete utilizzando un sistema software basato sulle classi del seguente diagramma UML



che possono essere così implementate in linguaggio Java[1]:

```
public class Antenna {
  private float latitudine;
  private float longitudine;
  private String tipo;

  // costruttore
  public Antenna(float latitudine, float longitudine,
                 String tipo) {
    this.latitudine = latitudine;
    this.longitudine = longitudine;
    this.tipo = tipo;
  }
```

1. Nella realtà la classe *Registro* disporrà di metodi per salvare in modo permanente su file il registro delle antenne e per ripristinarlo a partire da un file di salvataggio.

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Vanuti © Zanichelli editore 2018

```
  // costruttore di copia
  public Antenna(Antenna antenna) {
    this.latitudine = antenna.latitudine;
    this.longitudine = antenna.longitudine;
    this.tipo = antenna.tipo;
  }

  // metodi getter
  public float getLatitudine() {
    return latitudine;
  }
  public float getLongitudine() {
    return longitudine;
  }
  public String getTipo() {
    return tipo;
  }

  // metodi setter
  public void setLatitudine(float latitudine) {
    this.latitudine = latitudine;
  }
  public void setLongitudine(float longitudine) {
    this.longitudine = longitudine;
  }
  public void setTipo(String tipo) {
    this.tipo = tipo;
  }
}

public class Registro {
  public static final int NUMERO_MASSIMO_ANTENNE = 1000000;
  private Antenna[] antenne;
  private int numero_antenne;

  // costruttore
  public Registro() {
    antenne = new Antenna[NUMERO_MASSIMO_ANTENNE];
    numero_antenne = 0;
  }

  // metodi per aggiungere una nuova antenna al registro
  // restituiscono il numero dell'antenna nel registro
  public int aggiungiAntenna(float latitudine,
                             float longitudine, String tipo) {
    Antenna antenna = new Antenna(latitudine, longitudine,
                                  tipo);
    antenne[numero_antenne] = antenna;
    numero_antenne++;
    return (numero_antenne-1);
  }
  public int aggiungiAntenna(Antenna antenna) {
    antenne[numero_antenne] = new Antenna(antenna);
    numero_antenne++;
    return (numero_antenne-1);
  }
```


Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

```
  // restituisce il numero di antenne nel registro
  public int getNumeroAntenne() {
    return numero_antenne;
  }

  // restituisce le informazioni relative a un'antenna
  // noto il numero
  public Antenna getAntenna(int numero_antenna) {
    if (numero_antenna > numero_antenne || numero_antenna < 0)
      return null;
    return new Antenna(antenne[numero_antenna]);
  }

  // modifica le informazioni relative a un'antenna
  // noto il numero
  public void modificaAntenna(int numero_antenna, String tipo,
                              float latitudine,
                              float longitudine) {
    if (numero_antenna > numero_antenne || numero_antenna < 0)
      return;
    antenne[numero_antenna].setLatitudine(latitudine);
    antenne[numero_antenna].setLongitudine(longitudine);
    antenne[numero_antenna].setTipo(tipo);
  }

  // conta il numero di antenne presenti
  // in una data area geografica
  public int contaAntenne(float latitudine_minima,
                          float latitudine_massima,
                          float longitudine_minima,
                          float longitudine_massima) {
    int conta_antenne = 0;

    for (int n=0; n<numero_antenne; n++)
      if (antenne[n].getLatitudine() < latitudine_massima &&
          antenne[n].getLatitudine() > latitudine_minima &&
          antenne[n].getLongitudine() < longitudine_massima &&
          antenne[n].getLongitudine() > longitudine_minima)
        conta_antenne++;

    return conta_antenne;
  }
}
```

Il metodo *contaAntenne* consente di sapere quante antenne sono presenti in un'area geografica delimitata da valori minimi/massimi di latitudine e longitudine, ma se il numero di antenne registrate è elevato la natura intrinsecamente sequenziale dell'algoritmo di ricerca comporta un tempo di esecuzione notevole. La tecnica del *multi-threading* consente di sfruttare i moderni processori *multi-core* eseguendo più ricerche in parallelo su sezioni distinte del vettore che implementa il registro delle antenne.

Il modo più semplice per la creazione di un *thread* – cioè di un flusso di esecuzione concorrente – nel linguaggio di programmazione Java con-

Multi-threading

Per *multi-threading* si intende la possibilità di avere flussi di esecuzione paralleli e possibilmente contemporanei nel contesto di un unico processo. Un *thread* è una funzione – o un metodo in un contesto OO – la cui esecuzione avviene disponendo di una memoria locale privata e, se possibile, contemporaneamente agli altri *thread* del processo. Oltre a garantire – almeno con i moderni processori *multi-core* – maggiori prestazioni e una più semplice progettazione dell'architettura software di alcuni tipi di applicazioni, il *multi-threading* espone gli attributi delle classi a una possibile corruzione dovuta ad accessi non sincronizzati: per trarre vantaggio dalla potenza del *multi-threading* è necessario adottare tecniche di programmazione specifiche che evitano la corruzione dei dati utilizzati in modo concorrente da più *thread*.

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Vanuti © Zanichelli editore 2018

siste nell'istanziare una classe che deriva dalla classe *Thread* del *package* *java.lang*: è necessario ridefinire un metodo avente la seguente firma

```
public void run();
```

il cui corpo è il codice eseguito dal *thread*. La classe *Thread* definisce un metodo *start* la cui invocazione comporta l'effettiva creazione del *thread* e l'esecuzione del codice definito dal metodo *run*.

La seguente classe implementa un *thread* che effettua il conteggio delle antenne, anziché sull'intero vettore, su una specifica sezione del registro:

```
public class Ricerca extends java.lang.Thread {
  private Registro registro;
  private int numero_antenna_minimo;
  private int numero_antenna_massimo;
  private float latitudine_minima;
  private float latitudine_massima;
  private float longitudine_minima;
  private float longitudine_massima;
  private int conta_antenne;

  // costruttore: inizializzazione dei parametri di ricerca
  public Ricerca(Registro registro, int numero_antenna_minimo,
                 int numero_antenna_massimo, float latitudine_minima,
                 float latitudine_massima, float longitudine_minima,
                 float longitudine_massima) {
    this.registro = registro;
    if (numero_antenna_minimo < 0 ||
        numero_antenna_minimo > Registro.NUMERO_MASSIMO_ANTENNE)
      this.numero_antenna_minimo = 0;
    else
      this.numero_antenna_minimo = numero_antenna_minimo;
    if (numero_antenna_massimo < 0 ||
        numero_antenna_massimo > Registro.NUMERO_MASSIMO_ANTENNE)
      this.numero_antenna_massimo = 0;
    else
      this.numero_antenna_massimo = numero_antenna_massimo;
    this.latitudine_minima = latitudine_minima;
    this.latitudine_massima = latitudine_massima;
    this.longitudine_minima = longitudine_minima;
    this.longitudine_massima = longitudine_massima;
    conta_antenne = 0;
  }

  // metodo "run": codice del thread
  public void run() {
    for (int n=numero_antenna_minimo; n<numero_antenna_massimo; n++)
      {
        Antenna antenna = registro.getAntenna(n);
        if (antenna.getLatitudine() < latitudine_massima &&
            antenna.getLatitudine() > latitudine_minima &&
            antenna.getLongitudine() < longitudine_massima &&
            antenna.getLongitudine() > longitudine_minima)
```


Questo file costituisce un'estensione online dei corsi di Fornichi, Meini, Veguti © Zanichelli editore 2018

```
            conta_antenne++;
        }
    }

    // restituisce il risultato della ricerca
    public int getConteggioAntenne() {
      return conta_antenne;
    }
}
```

Dato che il metodo *start* non prevede parametri da passare al *thread*, è il costruttore che si fa carico di memorizzare i parametri necessari per l'esecuzione negli attributi della classe che sono accessibili al codice del metodo *run*; inoltre, dato che non è previsto un valore di ritorno per il risultato, questo viene memorizzato in un attributo per il quale esiste uno specifico metodo *get*.

ESEMPIO Il seguente frammento di codice mostra come utilizzare la classe *Ricerca* per effettuare in parallelo la ricerca delle antenne comprese in una specifica area geografica, rispettivamente nella prima metà e nella seconda metà del registro, raddoppiando potenzialmente le prestazioni dell'algoritmo[2]:

```
...
int numero_antenne_totale, numero_antenne_trovate;
Registro registro;
...
...
...
numero_antenne_totale = registro.getNumeroAntenne();
Ricerca ricerca1 = new Ricerca(registro, 0,
                               numero_antenne_totale/2,
                               43.5, 43.6, 10.3, 10.4);
Ricerca ricerca2 = new Ricerca(registro,
                               numero_antenne_totale/2,
                               numero_antenne_totale, 43.5,
                               43.6, 10.3, 10.4);
ricerca1.start();
ricerca2.start();
while (ricerca1.isAlive() || ricerca2.isAlive());
numero_antenne_trovate = ricerca1.getConteggioAntenne() +
                         ricerca2.getConteggioAntenne();
...
```

Non è difficile riscrivere il codice dell'esempio in modo che la ricerca sia effettuata da più di due *thread* in parallelo, ma – dal punto di vista prestazionale – è del tutto inutile superare il numero di *core* del processore disponibili per l'esecuzione.

2. In realtà le prestazioni aumentano solo nel caso che siano disponibili *core* del processore per l'esecuzione indipendente dei *thread*; anche in questo caso il tempo di esecuzione non sarà in ogni caso la metà di quello di una ricerca sequenziale perché molti fattori contribuiscono a limitare l'incremento di prestazioni di un algoritmo parallelo rispetto alla versione sequenziale, questo risultato è noto come legge di Amdahl.

L'invocazione del metodo *start* restituisce immediatamente il controllo anche nel caso che l'esecuzione del metodo *run*, il cui corpo costituisce il codice eseguito dal *thread*, sia molto lunga, o potenzialmente infinita, consentendo al *thread* principale di proseguire in modo indipendente la propria esecuzione.

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Vanuti © Zanichelli editore 2018

za della restituzione del controllo da parte del metodo *System.in.read* – avviene mentre il *thread* esegue il metodo *sleep* viene generata un'eccezione di tipo *InterruptedException* la cui gestione mediante l'istruzione **break** prevede l'uscita dal ciclo di ripetizione della visualizzazione delle stringhe e la conseguente terminazione del metodo *run* e del *thread* stesso.

Dato che tutti i *thread* attivi concorrono per essere eseguiti utilizzando le stesse risorse, nel caso che due o più *thread* eseguiti contemporaneamente non svolgano lo stesso compito è possibile indicare alla macchina virtuale Java la diversa urgenza o importanza di ciascuno di essi in modo che sia dedicato un tempo di esecuzione maggiore al o ai *thread* con priorità maggiore e un tempo di esecuzione minore al o ai *thread* con priorità minore. Il metodo *setPriority* della classe *Thread* ha esattamente questo scopo: per ragioni di portabilità del codice tra macchine virtuali per piattaforme di esecuzione diverse è consigliato limitarsi ad assegnare una delle priorità predefinite: MIN_PRIORITY, NORM_PRIORITY o MAX_PRIORITY. Indipendentemente dalla priorità impostata un *thread* il cui codice raggiunge un punto di esecuzione per cui non risulta utile l'immediata prosecuzione dovrebbe invocare il metodo statico *yield* della classe *Thread* in modo da rilasciare momentaneamente risorse utili per l'esecuzione di un altro *thread*.

ESEMPIO

Il metodo *run* della classe *Ricerca* può essere modificato in modo da consentire l'esecuzione di altri *thread* ogni 1000 iterazioni:

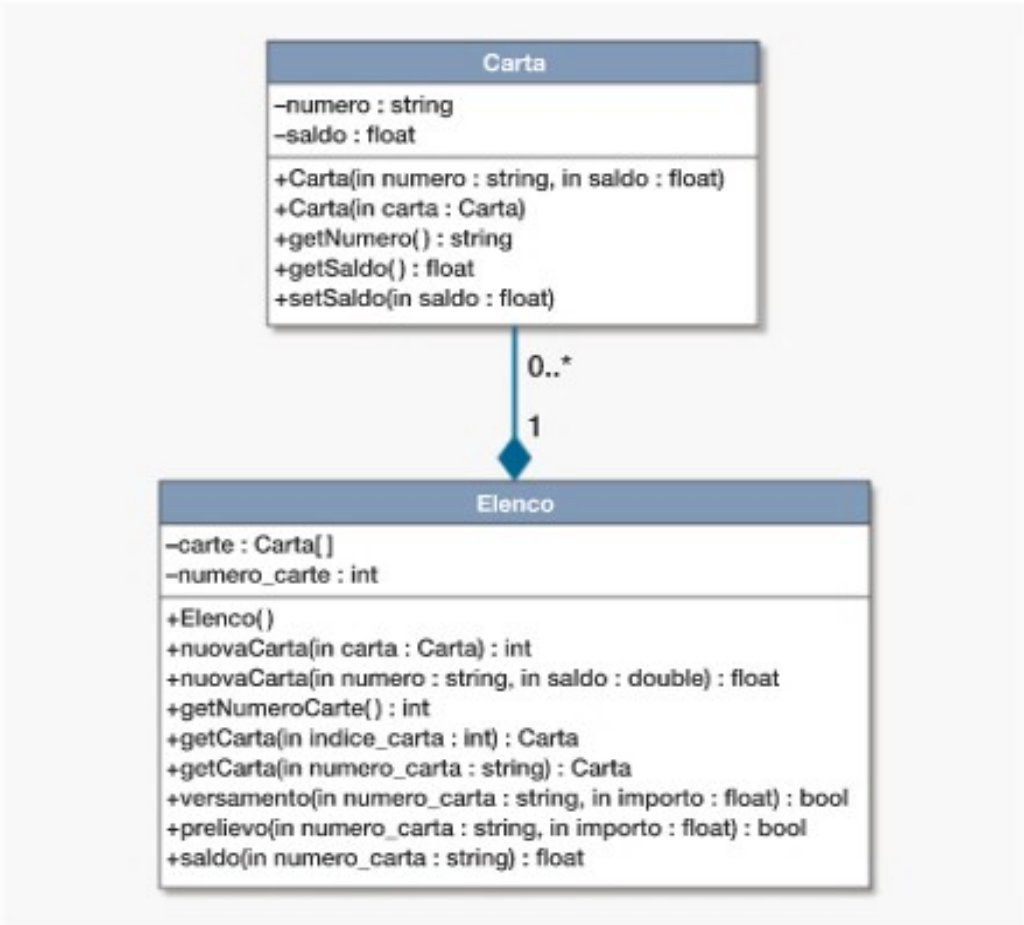
```
public void run() {
   for (int n=numero_antenna_minimo; n<numero_antenna_massimo;
        n++)
      {
        if (Thread.interrupted())
          return;
        if (n%1000 == 0)
          Thread.yield();
        Antenna antenna = registro.getAntenna(n);
        if (antenna.getLatitudine() < latitudine_massima &&
            antenna.getLatitudine() > latitudine_minima &&
            antenna.getLongitudine() < longitudine_massima &&
            antenna.getLongitudine() > longitudine_minima)
          conta_antenne++;
      }
  }
```

2 Condivisione di risorse tra *thread*

Un ente che emette carte di credito necessita di verificare il saldo di una carta ogni volta che questa viene impiegata per un'operazione di acquisto: a questo scopo è possibile ipotizzare una soluzione basata sulla seguente coppia di classi[3]

3. Una soluzione software reale dovrà necessariamente prevedere una struttura dati più efficiente di un *array*, come per esempio una tabella *hash*, per l'elenco delle carte emesse.

Questo file costituisce un'estensione online dei corsi di Fornichi, Meini, Veguti © Zanichelli editore 2018



che hanno la seguente implementazione in linguaggio Java:

```
public class Carta {
  String numero;
  float saldo;

  // costruttore
  public Carta(String numero, float saldo) {
    this.numero = numero;
    this.saldo = saldo;
  }

  // costruttore di copia
  public Carta(Carta carta) {
    this.numero = carta.numero;
    this.saldo = carta.saldo;
  }

  // metodi getter
  public String getNumero() {
    return numero;
  }
  public float getSaldo() {
    return saldo;
  }

  // metodi setter
  public void setSaldo(float saldo) {
    this.saldo = saldo;
  }
}
```


Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

```java
public class Elenco {
  public static final int NUMERO_MASSIMO_CARTE = 1000000;
  private Carta[] carte;
  private int numero_carte;

  // ricerca sequenziale di una carta dato il numero
  private int cercaCarta(String numero_carta) {
    for (int indice=0; indice<numero_carte; indice++)
      if (carte[indice].getNumero().equals(numero_carta))
        return indice;
    return -1; // carta non trovata
  }

  // costruttore
  public Elenco() {
    carte = new Carta[NUMERO_MASSIMO_CARTE];
    numero_carte = 0;
  }

  // metodi per aggiungere una nuova carta all'elenco
  // restituiscono il numero della carta nell'elenco
  public int nuovaCarta(Carta carta) {
    carte[numero_carte] = new Carta(carta);
    numero_carte++;
    return (numero_carte-1);
  }

  public int nuovaCarta(String numero, float saldo) {
    Carta carta = new Carta(numero, saldo);
    carte[numero_carte] = carta;
    numero_carte++;
    return (numero_carte-1);
  }

  // restituisce il numero di carte nell'elenco
  public int getNumeroCarte() {
    return numero_carte;
  }

  // restituisce le informazioni relative a una carta
  // noto l'indice
  public Carta getCarta(int numero_carta) {
    if (numero_carta > numero_carte)
      return null;
    return new Carta(carte[numero_carta]);
  }

  // restituisce le informazioni relative a una carta
  // noto il numero
  public Carta getCarta(String numero_carta) {
    int indice = cercaCarta(numero_carta);

    if (indice < 0)
      return null;
    else
      return new Carta(carte[indice]);
  }
```


Questo file costituisce un'estensione online dei corsi di Fogarichi, Meini, Venuti © Zanichelli editore 2018

```
  // ricarica di un importo su una carta
  public boolean ricarica(String numero_carta, float importo) {
    int indice = cercaCarta(numero_carta);

    if (indice < 0)
      return false;
    else
      {
        float saldo = carte[indice].getSaldo();
        saldo = saldo + importo;
        carte[indice].setSaldo(saldo);
        return true;
      }
  }

  // pagamento di un importo da una carta
  public boolean pagamento(String numero_carta, float importo) {
    int indice = cercaCarta(numero_carta);

    if (indice < 0)
      return false;
    else
      {
        float saldo = carte[indice].getSaldo();
        saldo = saldo - importo;
        carte[indice].setSaldo(saldo);
        return true;
      }
  }

  // richiesta del saldo di una carta
  float saldo(String numero_carta) {
    int indice = cercaCarta(numero_carta);

    if (indice < 0)
      return 0; // saldo fittizio per carta inesistente
    else
      return carte[indice].getSaldo();
  }
}
```

Ipotizzando che ricariche e pagamenti siano effettuati da *thread* distinti non è da escludere l'accesso concorrente e contemporaneo di più *thread* ai dati relativi a un'unica carta: in questo caso è possibile che il valore del saldo venga corrotto rendendo il sistema di fatto inservibile. La corruzione del valore del saldo può avvenire se questo viene modificato contemporaneamente da parte di due o più *thread*. Lo schema temporale di esecuzione delle singole istruzioni che segue illustra una condizione di corruzione del valore del saldo di 1000 € a fronte di una ricarica di 100 € e di un pagamento di 100 € che lo dovrebbero ovviamente lasciare inalterato.

Ricarica di 100 €	Pagamento di 100 €
`saldo = carte[indice].getSaldo();`	`saldo = carte[indice].getSaldo();`
`saldo = saldo + 100;`	`saldo = saldo - 100;`
`carte[indice].setSaldo(saldo);`	`carte[indice].setSaldo(saldo);`

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

Entrambi i *thread* acquisiscono un saldo iniziale pari a 1000 €: il *thread* che esegue il metodo di ricarica memorizza un saldo di 1100 € per la carta, ma immediatamente dopo il *thread* che esegue il codice di pagamento memorizza un saldo di 900 € con una differenza negativa di 100 € rispetto al valore corretto; una diversa sequenza temporale di esecuzione concorrente avrebbe memorizzato un saldo di 1100 € con una differenza positiva di 100 € rispetto al valore corretto.

La garanzia che il valore di una variabile a cui accedono più *thread* distinti non venga corrotto si ha solo se le operazioni sulla variabile che ne memorizza il valore sono eseguite in modo mutuamente esclusivo, impedendo la modifica concorrente da parte di metodi eseguiti contemporaneamente da *thread* distinti. La parola chiave **synchronized** del linguaggio Java deve essere premessa ai metodi «sincronizzati» di una classe la cui esecuzione deve essere cioè mutuamente esclusiva rispetto ai *thread* concorrenti: l'invocazione di un metodo sincronizzato da parte di un *thread* mentre un *thread* distinto non ha ancora terminato l'esecuzione dello stesso, o di un diverso, metodo sincronizzato comporta:

- l'attesa da parte del secondo *thread* fino a che il *thread* precedente non ha terminato l'esecuzione del metodo della classe (atomicità dell'esecuzione dei metodi della classe rispetto a invocazioni concorrenti);
- la visibilità da parte del codice eseguito dal secondo *thread* dell'eventuale aggiornamento dei valori degli attributi della classe da parte del codice del *thread* precedente.

Monitor

Il meccanismo di sincronizzazione dei *thread* concorrenti del linguaggio Java è noto come *monitor*. Un monitor è infatti un oggetto i cui metodi possono essere eseguiti solo in modalità mutuamente esclusiva da parte di più *thread* concorrenti.
I monitor prevedono inoltre meccanismi di segnalazione simili a quelli resi disponibili dal linguaggio Java.

ESEMPIO

La versione che segue della classe *Elenco*, in cui tutti i metodi pubblici sono definiti utilizzando la parola chiave **synchronized**, garantisce l'atomicità di esecuzione dei metodi della classe da parte di *thread* distinti e, di conseguenza, l'integrità dei dati da essi modificati:

```
public class Elenco {
  public static final int NUMERO_MASSIMO_CARTE = 1000000;
  private Carta[] carte;
  private int numero_carte;

  // ricerca sequenziale di una carta dato il numero
  private int cercaCarta(String numero_carta) {
    for (int indice=0; indice<numero_carte; indice++)
      if (carte[indice].getNumero().equals(numero_carta))
        return indice;
    return -1; // carta non trovata
  }

  // costruttore
  public Elenco() {
    carte = new Carta[NUMERO_MASSIMO_CARTE];
    numero_carte = 0;
  }

  // metodi per aggiungere una nuova carta all'elenco
  // restituiscono il numero della carta nell'elenco
```


Questo file costituisce un'estensione online dei corsi di Fornichi, Meini, Vegnuti © Zanichelli editore 2018

```
    synchronized public int nuovaCarta(Carta carta) {
      carte[numero_carte] = new Carta(carta);
      numero_carte++;
      return (numero_carte-1);
    }

    synchronized public int nuovaCarta(String numero, float saldo) {
      Carta carta = new Carta(numero, saldo);
      carte[numero_carte] = carta;
      numero_carte++;
      return (numero_carte-1);
    }

    // restituisce il numero di carte nell'elenco
    synchronized public int getNumeroCarte() {
      return numero_carte;
    }

    // restituisce le informazioni relative a una carta noto
    // l'indice
    synchronized public Carta getCarta(int numero_carta) {
      if (numero_carta > numero_carte)
        return null;
      return new Carta(carte[numero_carta]);
    }

    // restituisce le informazioni relative a una carta noto
    // il numero
    synchronized public Carta getCarta(String numero_carta) {
      int indice = cercaCarta(numero_carta);

    if (indice < 0)
      return null;
    else
      return new Carta(carte[indice]);
    }

    // ricarica di un importo su una carta
    synchronized public boolean ricarica(String numero_carta,
                                         float importo) {
      int indice = cercaCarta(numero_carta);

      if (indice < 0)
        return false;
      else
        {
          float saldo = carte[indice].getSaldo();
          saldo = saldo + importo;
          carte[indice].setSaldo(saldo);
          return true;
        }
    }

    // pagamento di un importo da una carta
    synchronized public boolean pagamento(String numero_carta,
                                          float importo) {
      int indice = cercaCarta(numero_carta);
      if (indice < 0)
```


Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Vanuti © Zanichelli editore 2018

```
        return false;
      else
          {
            float saldo = carte[indice].getSaldo();
            saldo = saldo - importo;
            carte[indice].setSaldo(saldo);
            return true;
          }
    }

    // richiesta del saldo di una carta
    synchronized float saldo(String numero_carta) {
      int indice = cercaCarta(numero_carta);

      if (indice < 0)
        return 0; // saldo fittizio per carta inesistente
      else
        return carte[indice].getSaldo();
    }
  }
```

Il costruttore di una classe non può essere invocato da più *thread* relativamente allo stesso oggetto per cui non può essere definito di tipo **synchronized**. Un metodo privato che viene invocato esclusivamente da metodi sincronizzati – come il metodo *cercaCarta* dell'esempio precedente – non è necessario che sia esso stesso definito di tipo **synchronized**.

In alcune situazioni rendere sincronizzati tutti i metodi di una classe non è sufficiente a garantire la correttezza dell'invocazione in successione dei metodi rispetto a eventuali invocazioni concorrenti da parte di altri *thread*.

Collezioni sincronizzate

L'ambiente di programmazione del linguaggio Java rende disponibili classi «collezione» sincronizzate il cui uso è raccomandato in un contesto *multi-thread*. Come le collezioni standard del linguaggio, le collezioni sincronizzate comprendono liste, code, tabelle *hash*, alberi, ecc.

ESEMPIO

Volendo gestire carte prepagate è necessario impedire che il saldo divenga negativo mediante un codice simile al seguente:

```
Elenco carte_prepagate = new Elenco();
float importo_pagamento;
String numero_carta;
...
...
...
if (carte_prepagate.saldo(numero_carta) <= importo_pagamento)
  carte_prepagate.pagamento(numero_carta, importo_pagamento);
```

Nel caso che l'esecuzione concorrente di un *thread* che invoca il metodo di modifica del saldo intervenga dopo la valutazione della condizione dell'istruzione condizionale il metodo *pagamento* potrebbe essere eseguito anche se la condizione che nel frattempo si è modificata non dovesse risultare più vera: è necessario che la valutazione della condizione e l'invocazione del metodo di pagamento avvengano in modo atomico.

L'istruzione **synchronized** del linguaggio Java consente di acquisire un *lock* su un oggetto in modo che l'esecuzione delle istruzioni del blocco dipendente dall'istruzione **synchronized** stessa risulti atomica rispetto all'esecuzione di blocchi di istruzioni dipendenti da altre istruzioni **synchronized** sullo stesso oggetto da parte di *thread* distinti.

Questo file costituisce un'estensione online dei corsi di Fornichi, Meini, Venuti © Zanichelli editore 2018

ESEMPIO Il modo corretto per gestire le carte prepagate impedendo che il saldo divenga negativo a causa di un'operazione di pagamento è il seguente:

```
Elenco carte_prepagate = new Elenco();
float importo_pagamento;
String numero_carta;
...
...
...
synchronized (carte_prepagate) {
  if (carte_prepagate.saldo(numero_carta) <= importo_pagamento)
    carte_prepagate.pagamento(numero_carta, importo_pagamento);
}
```

OSSERVAZIONE L'esecuzione del codice dei metodi di una classe definiti di tipo `synchronized` o delle istruzioni di un blocco dipendente da un'istruzione `synchronized` sullo stesso oggetto è atomica solo rispetto a invocazioni da parte di *thread* diversi; nel caso che l'invocazione avvenga da parte dello stesso *thread* è ammessa l'acquisizione «rientrante» del *lock*.

Un'istruzione `synchronized` può essere usata per migliorare le prestazioni dei metodi di una classe invocati da più *thread* concorrenti: invece di definire l'intero metodo come `synchronized`, impedendo di fatto l'esecuzione dei metodi della classe da parte di altri *thread*, è possibile imporre la sincronizzazione esclusivamente sull'accesso agli attributi di interesse e in condizioni potenzialmente critiche aumentando in misura notevole le prestazioni dei *thread* che invocano i metodi della classe.

ESEMPIO La classe che segue implementa un elenco di carte delle quali è possibile richiedere e aggiornare (mediante operazioni di ricarica e di pagamento) il saldo, ma che – una volta inizializzato con l'*array* di carte fornito come parametro al costruttore – non può essere esteso con l'aggiunta di nuove carte. Utilizzando l'istruzione viene sincronizzato l'accesso alla singola carta (uno specifico elemento dell'*array*) in modo da evitare corruzioni dovute all'interferenza tra *thread* che aggiornano/acquisiscono il saldo della stessa carta, ma consentendo a *thread* distinti di modificare in modo concorrente carte distinte (elementi diversi dell'*array*) aumentando notevolmente l'efficienza dell'invocazione dei metodi della classe.

```
public class ElencoRapido {
  private Carta[] carte;
  private int numero_carte;

  // ricerca sequenziale di una carta dato il numero
  private int cercaCarta(String numero_carta) {
    for (int indice=0; indice<numero_carte; indice++)
      if (carte[indice].getNumero().equals(numero_carta))
        return indice;
    return -1; // carta non trovata
  }

  // costruttore: copia dell'elenco di carte fornito come parametro
  public ElencoRapido(Carta[] elenco) {
    carte = new Carta[elenco.length];
    for (int indice=0; indice<numero_carte; indice++)
```

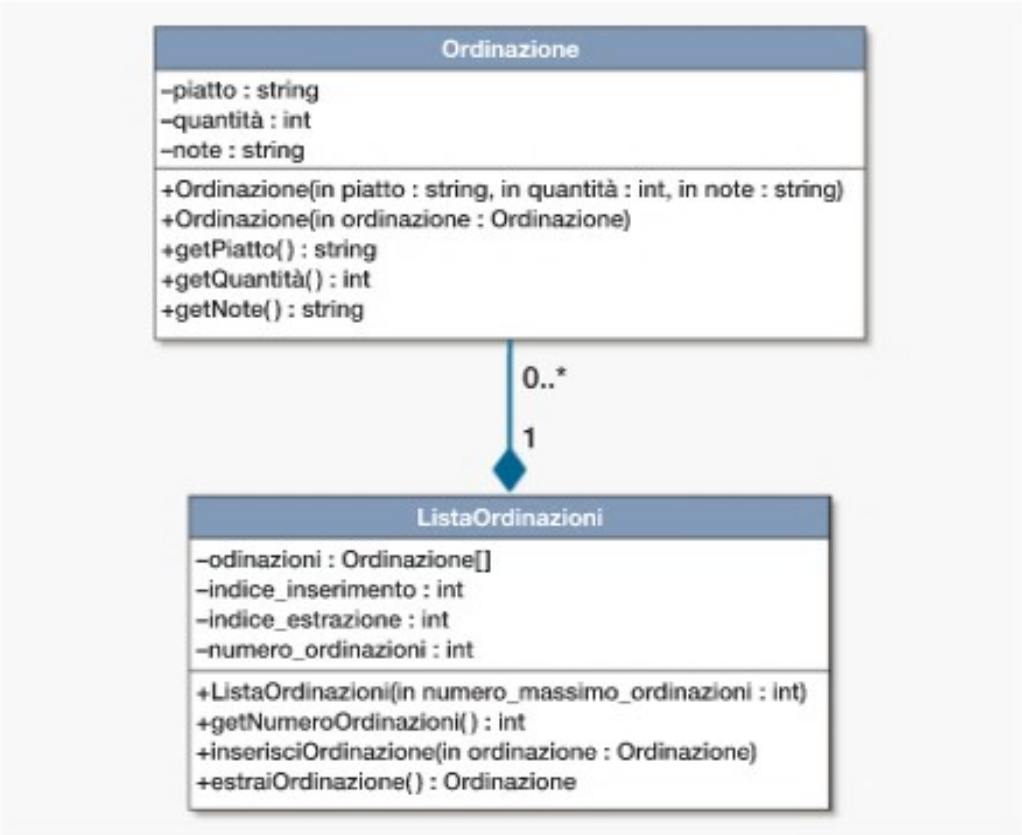

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Vanuti © Zanichelli editore 2018

```
        carte[indice] = new Carta(elenco[indice]);
      numero_carte = 0;
    }

    // restituisce il numero di carte
    public int getNumeroCarte() {
      return numero_carte;
    }

    // ricarica di un importo su una carta
    public boolean ricarica(String numero_carta, float importo) {
      int indice = cercaCarta(numero_carta);

      if (indice < 0)
        return false;
      else
        {
          synchronized (carte[indice]) {
            carte[indice].setSaldo(carte[indice].getSaldo()+importo);
          }
        return true;
        }
    }

    // pagamento di un importo da una carta
    public boolean pagamento(String numero_carta, float importo) {
      int indice = cercaCarta(numero_carta);

      if (indice < 0)
        return false;
      else
        {
          synchronized (carte[indice]) {
            carte[indice].setSaldo(carte[indice].getSaldo()-importo);
          }
          return true;
        }
    }

    // richiesta del saldo di una carta
    float saldo(String numero_carta) {
      float saldo_carta;
      int indice = cercaCarta(numero_carta);

      if (indice < 0)
        return 0; // saldo fittizio per carta inesistente
      else
        {
          synchronized (carte[indice]) {
            saldo_carta = carte[indice].getSaldo();
          }
          return saldo_carta;
        }
    }
  }
```


Questo file costituisce un'estensione online dei corsi di Fornichi, Meini, Vegnuti © Zanichelli editore 2018

3 Sincronizzazione dei *thread*

In un piccolo ristorante *fast food* le ordinazioni si presentano alla cassa posta all'ingresso: la cucina le prende in carico una alla volta, nell'ordine in cui sono state effettuate. Il cassiere predispone per ogni piatto ordinato un foglietto che ne riporta il nome, la quantità ordinata e le eventuali note di preparazione e lo pone in cima alla pila delle ordinazioni precedenti; il cuoco quando è disponibile prende il primo foglietto in fondo alla pila e ne inizia la preparazione. L'informatizzazione del processo può essere basata sulle seguenti classi:



La classe *ListaOrdinazioni* realizza una coda con politica FIFO (*First-In First-Out*) di oggetti di tipo *Ordinazione* impiegando un vettore e due indici, uno per la posizione in cui inserire la prossima ordinazione effettuata dalla cassa e l'altro per la posizione da cui estrarre la prossima ordinazione per la preparazione in cucina; entrambi gli indici sono incrementati calcolando come risultato il resto della divisione per la dimensione del vettore in modo da utilizzare le posizioni in modo ciclico[4]:

Anche se condividono la lista delle ordinazioni il cassiere e il cuoco svolgono attività indipendenti e sono quindi rappresentati da *thread* distinti: i metodi *inserisciOrdinazione* ed *estraiOrdinazione* sono di conseguenza invocati da *thread* concorrenti e devono essere sincronizzati. Una possibile implementazione in linguaggio Java delle classi *Ordinazione* e *ListaOrdinazioni* è la seguente.

Deadlock e starvation

Una scrittura non attenta del codice di sincronizzazione dei *thread* può comportare una situazione di stallo nota come *deadlock*: un *deadlock* avviene quando due *thread* sono ciascuno in attesa di una segnalazione da parte dell'altro e nessuno dei due è in grado di proseguire la computazione.
Una situazione diversa, ma ugualmente da evitare, è quella nota come *starvation*: in questa condizione un *thread* non è in grado di accedere a una risorsa perché altri *thread* – eventualmente aventi una priorità maggiore – la bloccano per periodi di tempo prolungati.

4. Questo classico esempio ha un valore esclusivamente didattico: il linguaggio Java dispone di classi specifiche per la gestione di liste con politica FIFO.

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Vanuti © Zanichelli editore 2018

```java
public class Ordinazione {
  private String piatto;
  private int quantità;
  private String note;

  // costruttore
  public Ordinazione(String piatto, int quantità, String note) {
    this.piatto = piatto;
    this.quantità = quantità;
    this.note = note;
  }

  // costruttore di copia
  public Ordinazione(Ordinazione ordinazione) {
    this.piatto = ordinazione.piatto;
    this.quantità = ordinazione.quantità;
    this.note = ordinazione.note;
  }

  // metodi getter
  public String getNote() {
    return note;
  }
  public String getPiatto() {
    return piatto;
  }
  public int getQuantità() {
    return quantità;
  }

  // conversione in stringa
  public String toString() {
    return piatto + " " + note + " " + quantità;
  }
}

public class ListaOrdinazioni {
  private Ordinazione[] ordinazioni;
  private volatile int indice_inserimento;
  private volatile int indice_estrazione;
  private volatile int numero_ordinazioni;

  // costruttore
  public ListaOrdinazioni(int numero_massimo_ordinazioni) {
    ordinazioni = new Ordinazione[numero_massimo_ordinazioni];
    indice_inserimento = 0;
    indice_estrazione = 0;
    numero_ordinazioni = 0;
  }

  // restituisce il numero delle ordinazioni in sospeso
  public int getNumeroOrdinazioni() {
    return numero_ordinazioni;
  }

  // inserimento di una nuova ordinazione nella lista
  public synchronized void inserisciOrdinazione
                            (Ordinazione ordinazione) {
```


Questo file costituisce un'estensione online dei corsi di Fogarichi, Meini, Vegnuti © Zanichelli editore 2018

```
    while (numero_ordinazioni >= ordinazioni.length);
    ordinazioni[indice_inserimento] = ordinazione;
    indice_inserimento = (indice_inserimento + 1) % ordinazioni.length;
    numero_ordinazioni++;
  }

  // estrazione di una ordinazione dalla lista
  public synchronized Ordinazione estraiOrdinazione() {
    while (numero_ordinazioni <= 0);
    Ordinazione ordinazione = ordinazioni[indice_estrazione];
    indice_estrazione = (indice_estrazione + 1) % ordinazioni.length;
    numero_ordinazioni--;
    return ordinazione;
  }
}
```

La parola chiave **volatile** specifica gli attributi che sono accessibili a più *thread* concorrenti: il compilatore del linguaggio Java evita le consuete ottimizzazioni prestazionali allo scopo di renderne atomico l'aggiornamento.

Le attese attive eseguite dai metodi *inserisciOrdinazione* ed *estraiOrdinazione*, rispettivamente nel caso di lista piena e di lista vuota mediante dei cicli di verifica della condizione, impiegano inutilmente il tempo di esecuzione del processore.

Il metodo *wait* che ogni classe Java eredita dalla classe *Object* consente di sospendere il *thread* corrente in attesa del verificarsi di una condizione che viene segnalata da parte di un altro *thread* invocando il metodo *notify*, anch'esso ereditato da *Object*.

L'attesa del metodo *wait* può essere interrotta, oltre che invocando il metodo *notify*, dall'interruzione del *thread*: in questo caso viene generata l'eccezione *InterruptedException* che deve necessariamente essere gestita.

Utilizzando i metodi *wait*/*notify* per la gestione della sincronizzazione tra i *thread* che invocano i metodi *inserisciOrdinazione* ed *estraiOrdinazione* l'implementazione della classe *ListaOrdinazioni* diviene la seguente[5]:

5. Si tratta in ogni caso di un esempio didattico: un risultato analogo lo si può ottenere istanziando un oggetto di classe *BlockingQueue*.

```
public class ListaOrdinazioni {
  private Ordinazione[] ordinazioni;
  private volatile int indice_inserimento;
  private volatile int indice_estrazione;
  private volatile int numero_ordinazioni;

  // costruttore
  public ListaOrdinazioni(int numero_massimo_ordinazioni) {
    ordinazioni = new Ordinazione[numero_massimo_ordinazioni];
    indice_inserimento = 0;
    indice_estrazione = 0;
    numero_ordinazioni = 0;
  }

  // restituisce il numero delle ordinazioni in sospeso
  public int getNumeroOrdinazioni() {
```


Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Vanuti © Zanichelli editore 2018

```
    return numero_ordinazioni;
  }

  // inserimento di una nuova ordinazione nella lista
  public synchronized void inserisciOrdinazione
                          (Ordinazione ordinazione) {
    while (numero_ordinazioni >= ordinazioni.length) {
      try {
           wait();
          }
      catch (InterruptedException eccezione) {
        return;
      }
    }
    ordinazioni[indice_inserimento] = ordinazione;
    indice_inserimento = (indice_inserimento + 1) % ordinazioni.length;
    numero_ordinazioni++;
    notify(); // risveglia thread di estrazione
  }

  // estrazione di una ordinazione dalla lista
  public synchronized Ordinazione estraiOrdinazione() {
    while (numero_ordinazioni <= 0) {
      try {
           wait();
          }
      catch (InterruptedException eccezione) {
        return null;
      }
    }
    Ordinazione ordinazione = ordinazioni[indice_estrazione];
    indice_estrazione = (indice_estrazione + 1) % ordinazioni.length;
    numero_ordinazioni--;
    notify(); // risveglia thread di inserimento
    return ordinazione;
  }
}
```

Per verificare il corretto funzionamento del codice della classe *ListaOrdinazioni* è necessario realizzare le classi che simulano il comportamento del cassiere e del cuoco[6]:

6. In entrambi i casi tra un'operazione e la successiva viene inserito un tempo di attesa casuale inferiore al secondo per simulare una situazione realistica.

```
public class Cassiere extends Thread {
  private Ordinazione ordinazioni[];
  private ListaOrdinazioni lista;

  public Cassiere(ListaOrdinazioni lista) {
    ordinazioni = new Ordinazione[5];
    ordinazioni[0] = new Ordinazione("Hamburger", 1, "senza cipolla");
    ordinazioni[1] = new Ordinazione("Hot-dog", 1, "solo senape");
    ordinazioni[2] = new Ordinazione("Patatine fritte", 1, maionese");
    ordinazioni[3] = new Ordinazione("Cheesburger", 1, "");
    ordinazioni[4] = new Ordinazione("Pollo fritto", 1,"");
    this.lista = lista;
  }
```


Questo file costituisce un'estensione online dei corsi di Fornichi, Meini, Vegnuti © Zanichelli editore 2018

```
  public void run() {
    while (!Thread.interrupted()) {
      int attesa = (int)(Math.random()*1000); // tempo di attesa casuale
      try {
           Thread.sleep(attesa);
          }
      catch (InterruptedException eccezione) {
        return;
      }
      int scelta = (int)(Math.random()*5); // scelta casuale
      lista.inserisciOrdinazione(ordinazioni[scelta]);
      System.out.println("Cassiere: " + ordinazioni[scelta]);
    }
  }
}

public class Cuoco extends Thread {
  private ListaOrdinazioni lista;

  public Cuoco(ListaOrdinazioni lista) {
    this.lista = lista;
  }

  public void run() {
    while (!Thread.interrupted()) {
      int attesa = (int)(Math.random()*1000); // tempo di attesa casuale
      try {
           Thread.sleep(attesa);
          }
      catch (InterruptedException eccezione) {
        return;
      }
      Ordinazione ordinazione = lista.estraiOrdinazione();
      System.out.println("Cuoco: " + ordinazione);
    }
  }
}
```

Il seguente metodo *main*, inserito in un'apposita classe di test, consente di verificare il corretto funzionamento della lista delle ordinazioni:

```
public class Test {
  public static void main(String args[]) throws java.io.IOException {
    ListaOrdinazioni lista = new ListaOrdinazioni(10);
    Cassiere cassiere = new Cassiere(lista);
    Cuoco cuoco = new Cuoco(lista);

    cassiere.start();
    cuoco.start();
    int c = System.in.read(); // attesa inserimento carattere
    cassiere.interrupt();
    try {
         cassiere.join();
        }
    catch(InterruptedException eccezione) {
    }
```


Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

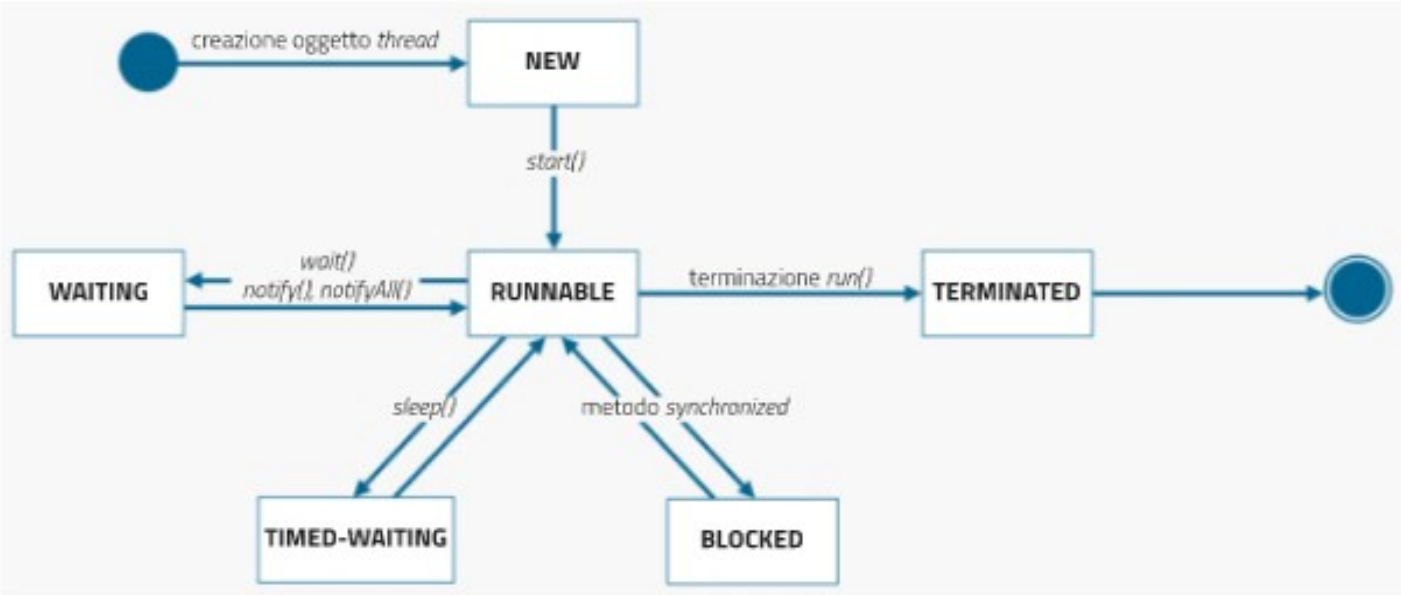
```
        cuoco.interrupt();
        try {
             cuoco.join();
            }
        catch(InterruptedException eccezione) {
        }
    }
}
```

Prevedendo di utilizzare la classe *ListaOrdinazioni* per un ristorante più grande, con più casse e più cuochi, i *thread* potenzialmente interessati a essere richiamati dallo stato di sospensione sono più di uno: in questo caso è preferibile utilizzare il metodo *notifyAll*, che segnala tutti i *thread* sospesi sull'oggetto, invece del metodo *notify* che segnala solo il primo *thread* che è sospeso sull'oggetto.

Concludiamo la trattazione della gestione della concorrenza nel linguaggio Java elencando gli stati in cui un *thread* Java può trovarsi:

- **NEW**: l'oggetto di classe *Thread* è stato creato, ma non è ancora in esecuzione;
- **RUNNABLE**: il *thread* è eseguibile (lo stato di esecuzione vera e propria viene schedulato dalla macchina virtuale e non è accessibile al codice del programma);
- **WAITING**: il *thread* si è autosospeso invocando il metodo *wait* in attesa di essere segnalato mediante invocazione del metodo *notify* o *notifyAll*;
- **TIMED-WAITING**; il *thread* si è sospeso per un periodo di tempo invocando il metodo *sleep*;
- **BLOCKED**: il *thread* è bloccato in attesa che il *lock* richiesto invocando un metodo sincronizzato o eseguendo un'istruzione `synchronized` sia rilasciato da un altro *thread*;
- **TERMINATED**: il *thread* è completato.

Il seguente diagramma UML di stato sintetizza gli stati in cui un *thread* può trovarsi:



Questo file costituisce un'estensione online dei corsi di Fornichi, Meini, Vegnuti © Zanichelli editore 2018

4 Uso di *thread pool*

Il *pattern* di progettazione software *thread pool* prevede di gestire le situazioni in cui è necessario, per motivi prestazionali, suddividere un compito tra più *thread* eseguiti contemporaneamente creando un numero adeguato, normalmente limitato, di *thread* a cui assegnare specifici *task* da svolgere.

Il *package java.util.concurrent* comprende la classe *Executors* che consente di istanziare un *thread pool* invocando i metodi statici riportati nella tabella che segue.

Metodo	Descrizione
`ExecutorService newSingleThreadExecutor()`	Restituisce un oggetto di tipo *ExecutorService* che impiega un solo *thread*: i *task* assegnati sono gestiti con una politica di tipo FIFO (*First-In First-Out*).
`ExecutorService newCachedThreadPool()`	Restituisce un oggetto di tipo *ExecutorService* che realizza un *thread pool* non limitato: per ogni *task* assegnato viene, se necessario, creato un nuovo *thread*.
`ExecutorService newFixedThreadPool(int N_thread)`	Restituisce un oggetto di tipo *ExecutorService* che realizza un *thread pool* limitato: ogni *task* viene assegnato a uno degli *N_thread* creati inizialmente.
`ExecutorService newWorkStealingPool()`	Restituisce un oggetto di tipo *ExecutorService* che realizza un *thread pool* limitato: il numero di *thread* a cui sono assegnati i *task* corrisponde al numero di *core* del processore di esecuzione.

L'interfaccia *ExecutorService* è implementata dalla classe *ThreadPoolExecutor*: gli oggetti restituiti dai metodi statici della classe *Executors* sono istanze di questa classe; la tabella che segue riporta i metodi fondamentali dell'interfaccia *ExecutorService*.

Metodo	Descrizione
`boolean isShutdown()`	Restituisce `true` se il *thread pool* è stato terminato.
`boolean awaitTermination(long timeout, TimeUnit unit)`	Attende, per il tempo massimo specificato, la terminazione di tutti i *task* del *thread pool* dopo l'invocazione del metodo *shutdown*; l'interruzione durante l'attesa solleva l'eccezione *InterruptedException*.
`boolean isTerminated()`	Restituisce `true` se, dopo la terminazione del *thread pool*, tutti i *task* sono effettivamente terminati.
`void shutdown()`	Inizia la terminazione del *thread pool* senza interrompere i *task* in esecuzione.
`Future<?> submit(Runnable task)`	Fornisce un nuovo *task* al *thread pool* nella forma di un oggetto istanza di una classe che implementa l'interfaccia *Runnable*; nel caso che il *task* non possa essere eseguito dal *thread pool* (per esempio perché terminato in precedenza invocando il metodo *shutdown*) solleva l'eccezione *RejectedExecutionException*.

OSSERVAZIONE L'istanza della classe generica *Future* restituita dall'invocazione del metodo *submit* permette di attendere la conclusione dell'esecuzione dello specifico *task* invocandone il metodo *get*, o di verificarla senza sospensione invocandone il metodo *isDone*.

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Vanuti © Zanichelli editore 2018

ESEMPIO

Utilizzando un *thread pool* è possibile effettuare la ricerca di un elemento in un vettore non ordinato generando un numero di *thread* uguale al numero di *core* del processore che esegue la JVM:

```
...
int numero_antenne_totale, numero_antenne_trovate;
Registro registro;
ExecutorService thread_pool;
...
...
...
// ottiene il numero di core del processore
int cores = Runtime.getRuntime().availableProcessors();
numero_antenne_totale = registro.getNumeroAntenne();
thread_pool = Executors.newWorkStealingPool();

for (int n=0; n<cores; n++) {
  Ricerca ricerca = new Ricerca(registro,
                                numero_antenne_totale/cores*n,
                                numero_antenne_totale/cores*(n+1),
                                43.5F, 43.6F, 10.3F, 10.4F);
  thread_pool.submit(ricerca);
}
thread_pool.shutdown();
try {
  // attende al massimo per 60 s la terminazione di tutti i thread
  thread_pool.awaitTermination(60, TimeUnit.SECONDS);
}
catch (InterruptedException eccezione) {
}
...
...
...
```


Questo file costituisce un'estensione online dei corsi di Fornichi, Meini, Vegnuti © Zanichelli editore 2018

I CONCETTI CHIAVE

***THREAD* IN JAVA**. Il modo più semplice per la creazione di un *thread* – cioè di un flusso di esecuzione concorrente – nel linguaggio di programmazione Java consiste nell'istanziare una classe che deriva dalla classe *Thread* del *package java.lang*: è necessario ridefinire il metodo *run* il cui corpo è il codice eseguito dal *thread*. La classe *Thread* definisce un metodo *start* la cui invocazione comporta l'effettiva creazione del *thread* e l'esecuzione del codice definito dal metodo *run*.

AVVIO DI UN *THREAD*. L'invocazione del metodo *start* restituisce immediatamente il controllo anche nel caso che l'esecuzione del metodo *run*, il cui corpo costituisce il codice eseguito dal *thread*, sia molto lunga, o potenzialmente infinita, consentendo al *thread* principale di proseguire in modo indipendente la propria esecuzione.

ATTESA DELLA TERMINAZIONE DI UN *THREAD*. Nel caso che il codice non abbia altro scopo che quello di attendere la terminazione di un *thread* è possibile invocare il metodo *join* che restituisce il controllo solo quando il *thread* è terminato.

USO INTERFACCIA *RUNNABLE*. La tecnica di creazione di un *thread* ereditando dalla classe *Thread* è però fortemente limitante: nel linguaggio Java non è infatti possibile ereditare da più di una classe, di conseguenza una classe che eredita dalla classe *Thread* non potrebbe ereditare da un'altra classe. È possibile ovviare a questo problema implementando l'interfaccia *Runnable*, anziché ereditare dalla classe *Thread*, che richiede la definizione del solo metodo *run*. L'esecuzione del *thread* in questo caso si ottiene istanziando un oggetto di classe *Thread* e fornendo al costruttore un riferimento a un oggetto istanza della classe che implementa l'interfaccia *Runnable* e invocando il metodo *start* dell'oggetto *thread*.

METODO *RUN*. La struttura del metodo *run* come un ciclo che si ripete fino a che il *thread* non viene interrotto è tipica della programmazione *multi-threading* in cui ogni *thread* svolge ripetutamente un compito: l'uscita dal ciclo è garantita dalla condizione di ripetizione basata sull'interrogazione dell'eventuale stato di interruzione.

METODI SINCRONIZZATI. La garanzia che il valore di una variabile condivisa tra più *thread* non venga corrotto si ha solo se le operazioni su di essa sono eseguite in modo mutuamente esclusivo, impedendo la modifica concorrente da parte di metodi eseguiti contemporaneamente da *thread* distinti. La parola chiave **`synchronized`** del linguaggio Java deve essere premessa ai metodi «sincronizzati» di una classe la cui esecuzione deve essere cioè mutuamente esclusiva rispetto ai *thread* concorrenti: l'invocazione di un metodo sincronizzato da parte di un *thread* mentre un *thread* distinto non ha ancora terminato l'esecuzione dello stesso, o di un diverso, metodo sincronizzato comporta: l'attesa da parte del secondo *thread* fino a che il *thread* precedente non ha terminato l'esecuzione del metodo della classe (atomicità dell'esecuzione dei metodi della classe rispetto a invocazioni concorrenti) e la visibilità da parte del codice eseguito dal secondo *thread* dell'eventuale aggiornamento dei valori degli attributi della classe da parte del codice del *thread* precedente.

BLOCCHI DI CODICE SINCRONIZZATI. L'istruzione **`synchronized`** del linguaggio Java consente di acquisire un *lock* su un oggetto in modo che l'esecuzione delle istruzioni del blocco dipendente dall'istruzione **`synchronized`** stessa risulti atomica rispetto all'esecuzione di blocchi di istruzioni dipendenti da altre istruzioni **`synchronized`** sullo stesso oggetto da parte di *thread* distinti.
Un'istruzione **`synchronized`** può essere usata per migliorare le prestazioni dei metodi di una classe invocati da più *thread* concorrenti: invece di definire l'intero metodo come **`synchronized`**, impedendo di fatto l'esecuzione dei metodi della classe da parte di altri *thread*, è possibile imporre la sincronizzazione esclusivamente sull'accesso agli attributi di interesse e in condizioni potenzialmente critiche aumentando in misura notevole le prestazioni dei *thread* che invocano i metodi della classe.

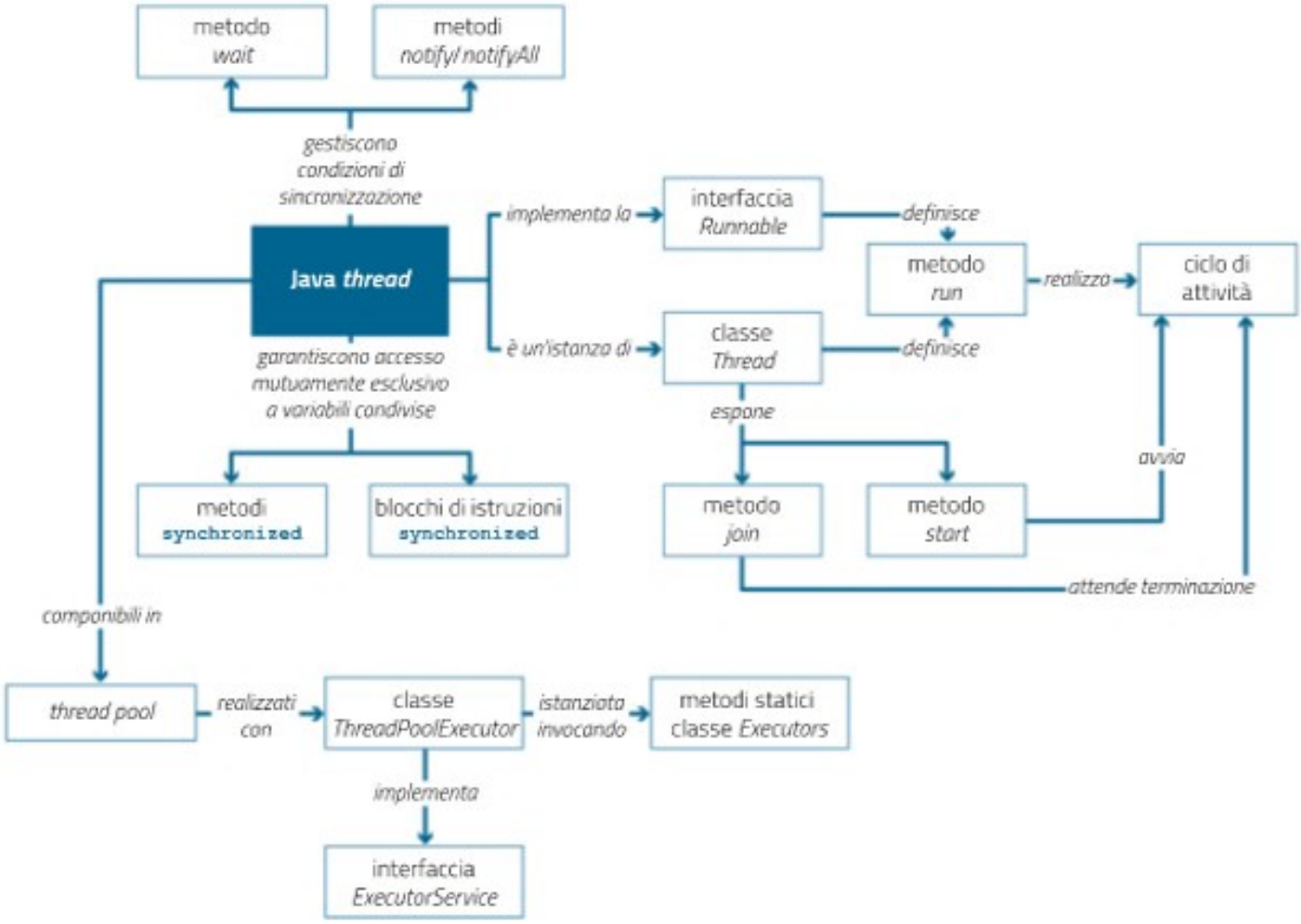
ATTESA E NOTIFICA DI CONDIZIONI. Il metodo *wait* che ogni classe Java eredita dalla classe *Object* consente di sospendere il *thread* corrente in attesa del verificarsi di una condizione che viene segnalata da parte di un altro *thread* invocando il metodo *notify* (o *notifyAll*), anch'esso ereditato da *Object*. Il metodo *notifyAll* segnala tutti i *thread* sospesi sull'oggetto, invece il metodo *notify* segnala solo il primo *thread* che è sospeso sull'oggetto.

THREAD POOL. Il *pattern* di progettazione software *thread pool* prevede di gestire le situazioni in cui è necessario, per motivi prestazionali, suddividere un compito tra più *thread* eseguiti contemporaneamente creando un numero adeguato, normalmente limitato, di *thread* a cui assegnare specifici *task* da svolgere.

CLASSE THREADPOOLEXECUTOR. Il *package java.util.concurrent* comprende la classe *Executors* che consente di istanziare un *thread pool* invocando specifici metodi statici che restituiscono un'istanza della classe *ThreadPoolExecutor* che implementa l'interfaccia *ExecutorService*.

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Vanuti © Zanichelli editore 2018

RIPASSA CON LA MAPPA



Questo file costituisce un'estensione online dei corsi di Fornichi, Meini, Venuti © Zanichelli editore 2018

A10

Ambiente di sviluppo *NetBeans* per linguaggio Java

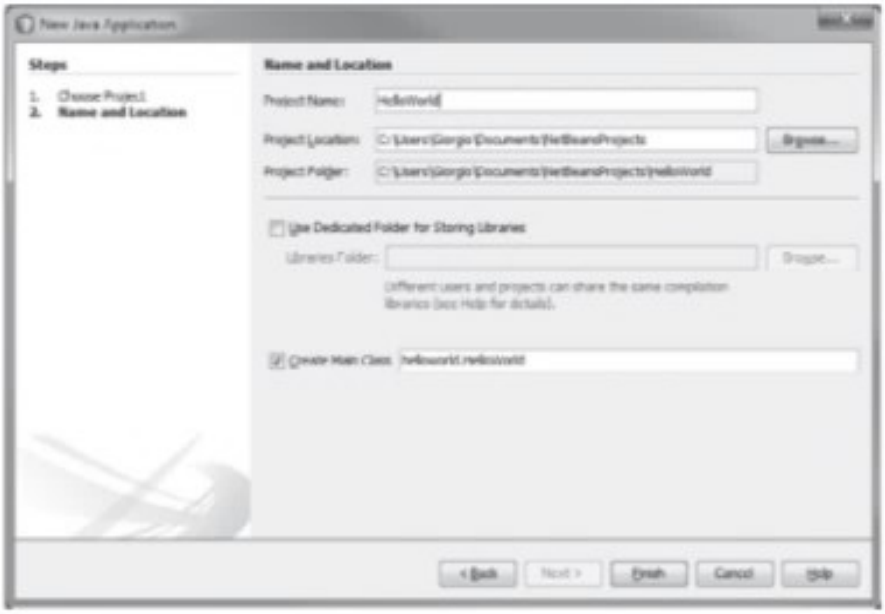
NetBeans è un IDE (*Integrated Development Environment*) multipiattaforma: può essere eseguito su Linux, Windows e MacOS e produrre codice eseguibile per questi sistemi operativi utilizzando i linguaggi di programmazione Java, C/C++ e molti altri. L'IDE *NetBeans* è stato inizialmente sviluppato presso l'Università di Praga negli ultimi anni del secolo scorso, ma nel 1999 fu acquisito da Sun Microsystems – la società che ha creato il linguaggio di programmazione Java – e successivamente mantenuto come progetto *open-source*. L'acquisizione di Sun Microsystems da parte di Oracle Corporation nel 2010 non ha generato discontinuità nella manutenzione ed evoluzione del progetto che nel 2016 è stato ceduto all'Apache Software Foundation, la fondazione che mantiene alcuni dei più importanti progetti software *open-source*. Il concetto fondamentale nel contesto di un IDE è quello di «progetto»: un progetto è un insieme strutturato di file sorgente la cui compilazione produce come risultato il *byte-code* eseguibile di un programma Java.

Nell'IDE *NetBeans* possono essere importati molti progetti diversi: è consigliabile impostare il progetto a cui si sta lavorando come «progetto principale» a cui si riferiscono implicitamente i comandi di compilazione ed esecuzione.

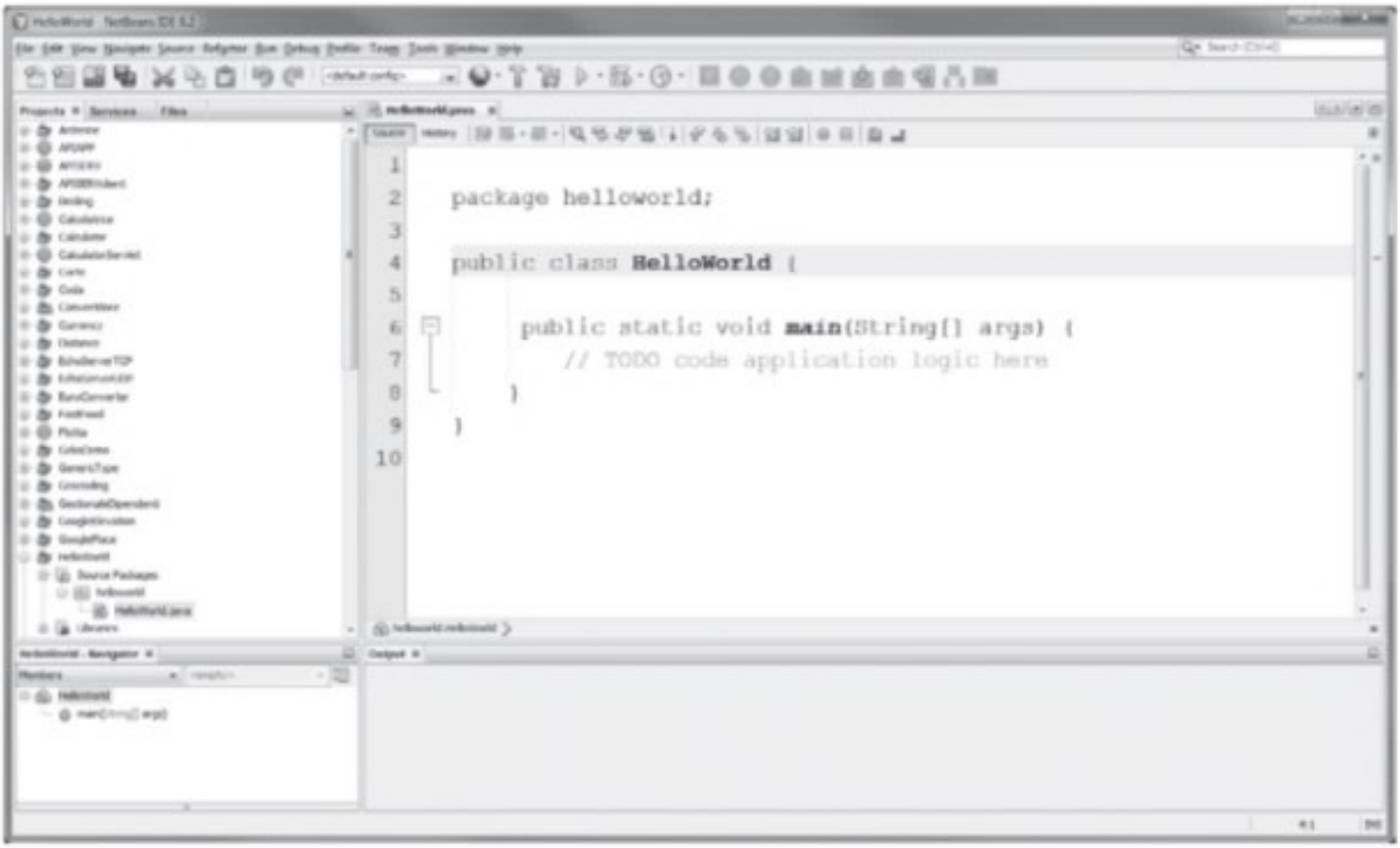
La creazione di un nuovo progetto in ambiente *NetBeans* si effettua con il comando *New Project* del menu *File* che attiva un *wizard* di semplice utilizzazione. La prima finestra richiede la selezione del tipo di progetto che nel caso più comune è *Java Application* della categoria *Java*:

La seconda finestra consente di denominare il progetto selezionando la directory di creazione e, opzionalmente, di indicare *package* e nome della classe principale.

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

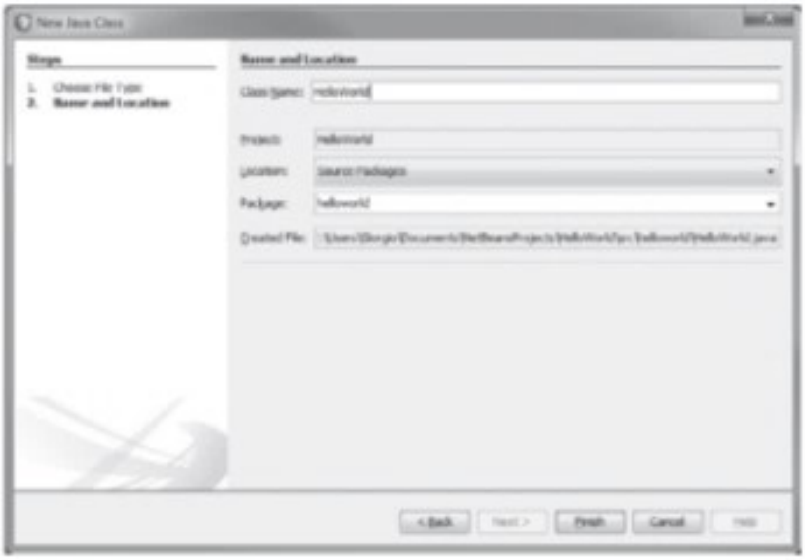


Il nuovo progetto viene aggiunto alla lista e selezionato automaticamente, se si è scelta la generazione automatica della classe principale è possibile iniziare a programmare:



Le convenzioni del linguaggio Java prevedono che ogni classe pubblica sia codificata in un file separato il cui nome coincide con quello della classe stessa: la creazione di nuove classi avviene selezionando la voce *Java class* del comando *New* del menu contestuale richiamato con il tasto destro del mouse sull'icona del *package*; il *wizard* di creazione della classe permette di definirne il nome. Nel caso che non sia stato definito un *package* ne esiste uno denominato *default package*.

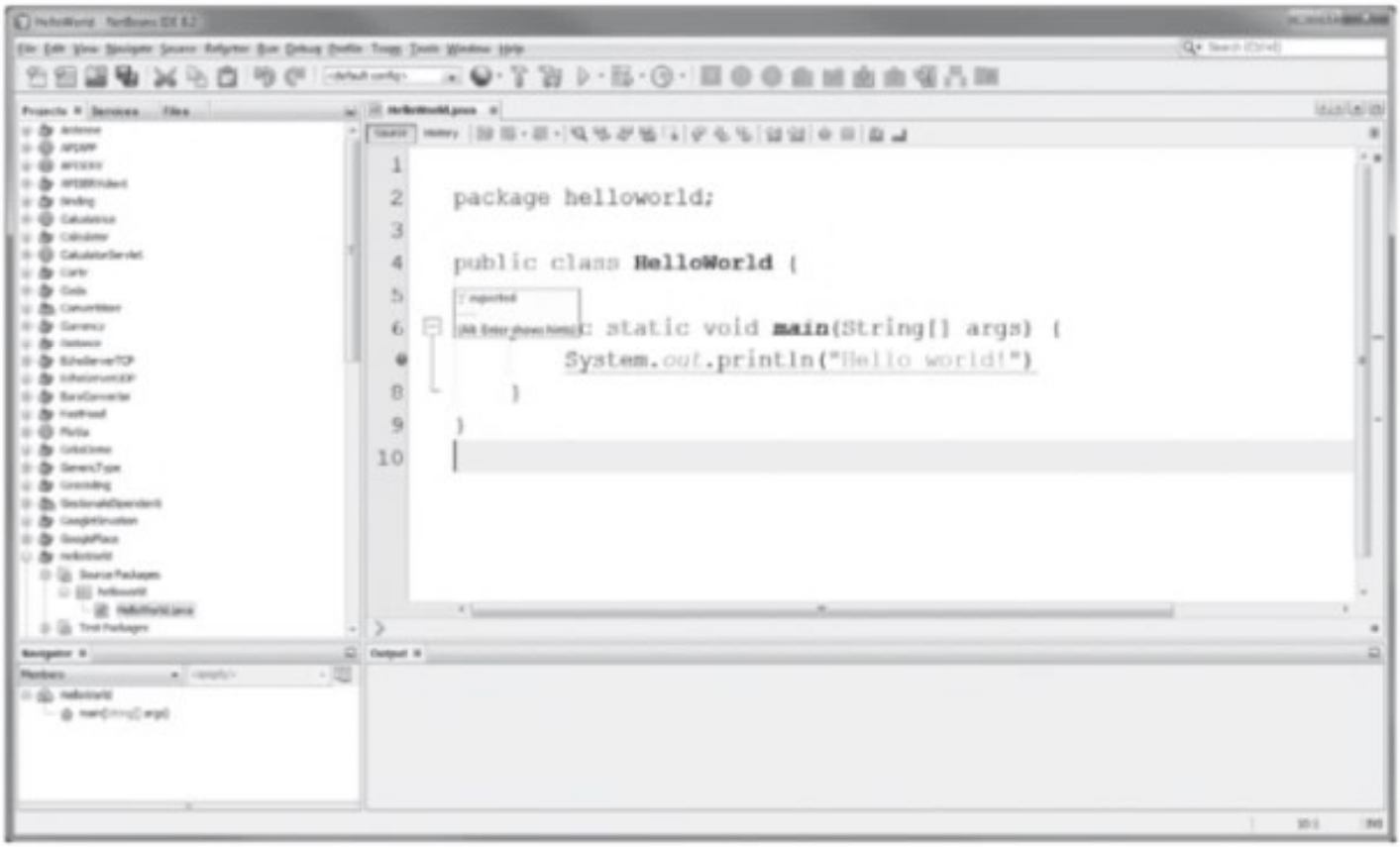
Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018



La documentazione del linguaggio e della piattaforma Java

Il sito https://docs.oracle.com/javase/ ospita la documentazione aggiornata relativa al linguaggio di programmazione Java e alle classi della piattaforma standard. In particolare l'accesso alla documentazione dell'API (*Application Program Interface*) in formato Javadoc risulta indispensabile nel corso dell'attività di programmazione.

L'IDE *NetBeans* segnala gli errori di codifica nel corso della digitazione del codice da parte del programmatore che può così verificare il tipo di errore posizionando il cursore del mouse sull'icona stessa dell'errore:



OSSERVAZIONE Nel corso della codifica risultano estremamente utili le funzionalità di generazione automatica del codice disponibili con il comando *Insert code* del menu *Source*: è infatti possibile generare automaticamente i costruttori standard e i metodi di accesso *get*/*set* a partire dalla definizione degli attributi di una classe.

Il salvataggio del codice avviene utilizzando l'icona dello specifico strumento, o i comandi dedicati del menu *File*; la compilazione del singolo file si avvia con il comando *Compile File* del menu contestuale che si ottiene

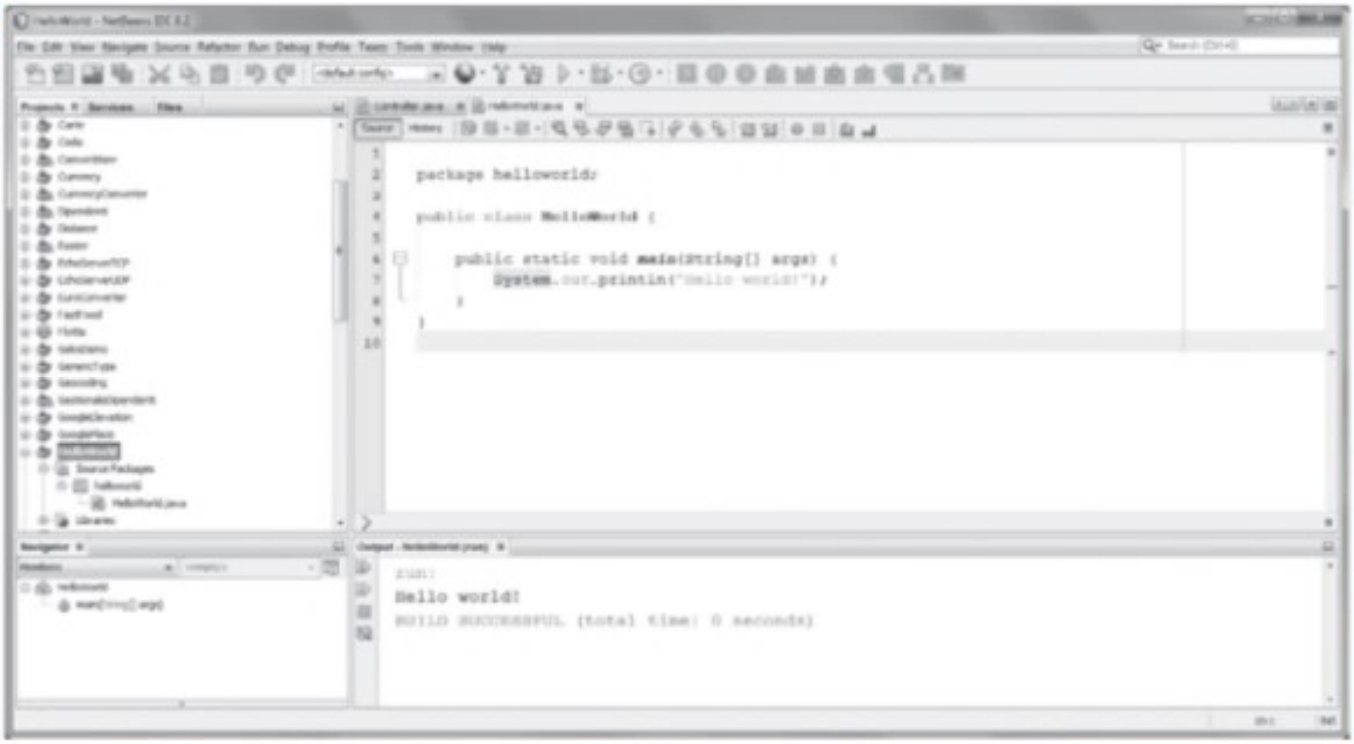
Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

selezionando con il tasto destro del mouse l'icona del file sorgente nell'area dei progetti. La compilazione di tutti i file di codice del progetto e la produzione del codice eseguibile in assenza di errori si realizza invece con il comando *Build Main Project* del menu *Run*, oppure con il comando *Build* del menu contestuale che si ottiene selezionando con il tasto destro del mouse l'icona del progetto[2].

Nella finestra di output viene in ogni caso visualizzato il risultato della compilazione e gli eventuali messaggi di errore che risultano essere collegamenti ipertestuali alla riga di codice sorgente che ha causato l'errore di compilazione in modo da agevolarne la correzione. Una volta compilato il programma senza errori è possibile provarne l'esecuzione direttamente dall'interno dell'IDE utilizzando il comando *Run Main Project* del menu *Run*, o il comando *Run* del menu contestuale del progetto, oppure attivando l'icona dello strumento specifico.

2 Nel menu contestuale sono presenti anche i comandi *Clean*, che elimina i file *byte-code* prodotti da compilazioni precedenti, e *Clean and Build*, che invoca automaticamente i comandi *Clean* e *Build* in sequenza.

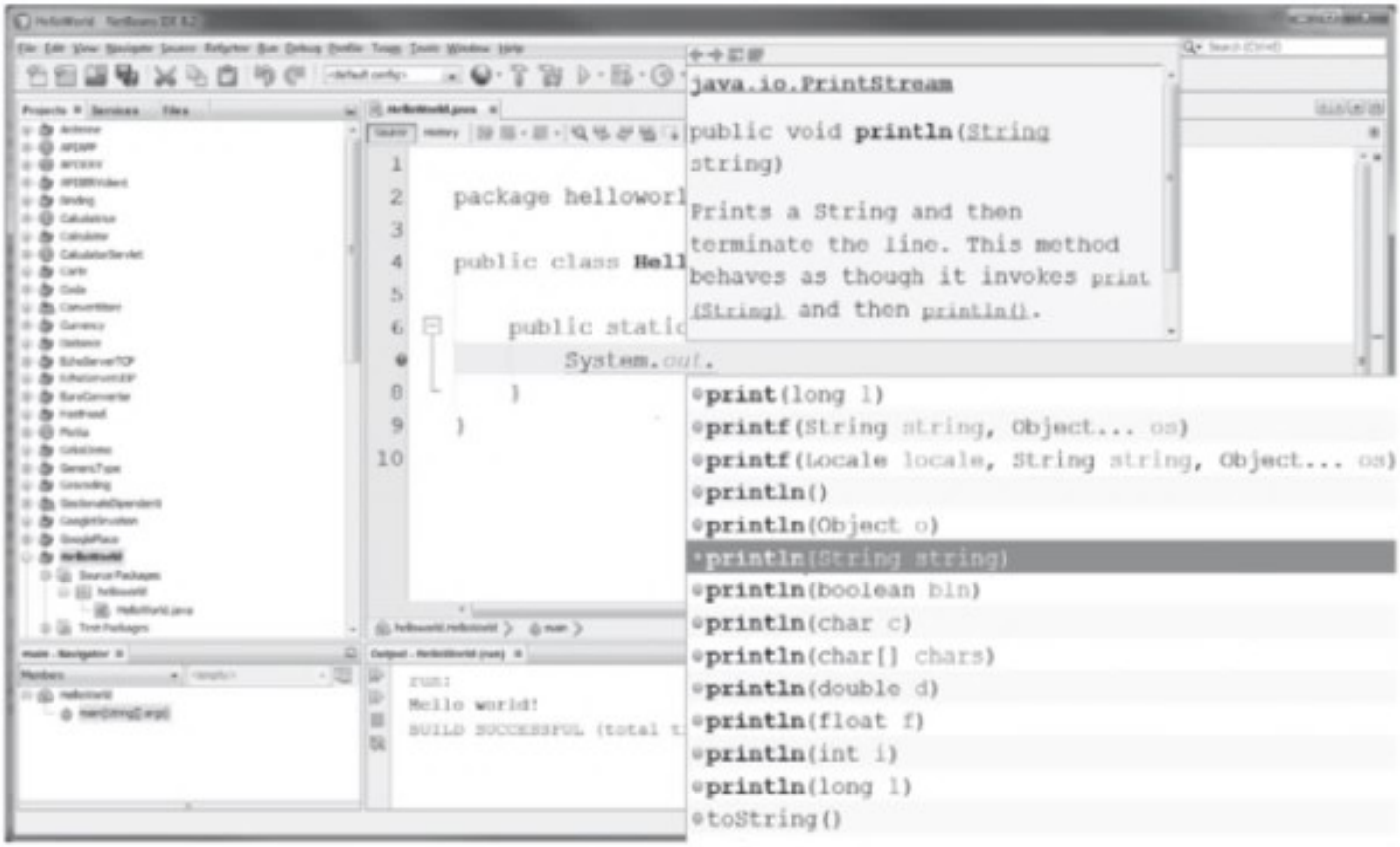
ESEMPIO L'output di un programma eseguito nel contesto dell'ambiente *NetBeans* viene visualizzato nella finestra di output:



OSSERVAZIONE Dato che ogni classe di un progetto Java può potenzialmente disporre di un metodo *main*, l'esecuzione si avvia dal metodo *main* della classe designata come classe principale del progetto: è possibile modificare la selezione della classe principale nelle proprietà del progetto, editabili selezionando il comando *Properties* del menu contestuale del progetto stesso.

Anche il programmatore più esperto necessita di consultare in continuazione la documentazione delle classi che utilizza per realizzare le proprie applicazioni: l'ambiente *NetBeans* visualizza automaticamente l'elenco completo dei metodi di una classe in fase di codifica e la documentazione dello specifico metodo che viene selezionato con il puntatore del mouse:

Questo file costituisce un'estensione online dei corsi di Fornichi, Meini, Venuti © Zanichelli editore 2018



OSSERVAZIONE La documentazione dei metodi delle classi e dei relativi parametri codificati dallo sviluppatore secondo lo standard *Javadoc* consente a *NetBeans* di visualizzarne il testo contestualmente all'uso nel codice digitato, esattamente nello stesso modo con cui visualizza la documentazione relativa ai metodi della classi della libreria del linguaggio Java.

1 *Debug* di programmi Java

Come in tutti gli IDE anche in *NetBeans* è possibile impostare «punti di interruzione» (*break-point*) in cui fermare l'esecuzione del codice per controllare il valore delle variabili allo scopo di effettuare il *debug* di un programma. Un punto di interruzione può essere impostato in corrispondenza di una linea di codice utilizzando il comando *Toggle Line Breakpoint* reso disponibile dal menu contestuale che si apre agendo con il pulsante destro del mouse nell'area di scrittura del codice, oppure eseguendo un semplice click del mouse in corrispondenza del numero di riga.

I punti di interruzione sono ignorati nel corso della normale esecuzione: per utilizzarli il programma deve essere eseguito con il comando *Debug Main Project* del menu *Debug*, o con il comando *Debug* del menu contestuale del progetto, oppure attivando l'icona dello specifico strumento.

ESEMPIO La seguente classe Java implementa la rappresentazione di un punto nel piano cartesiano:

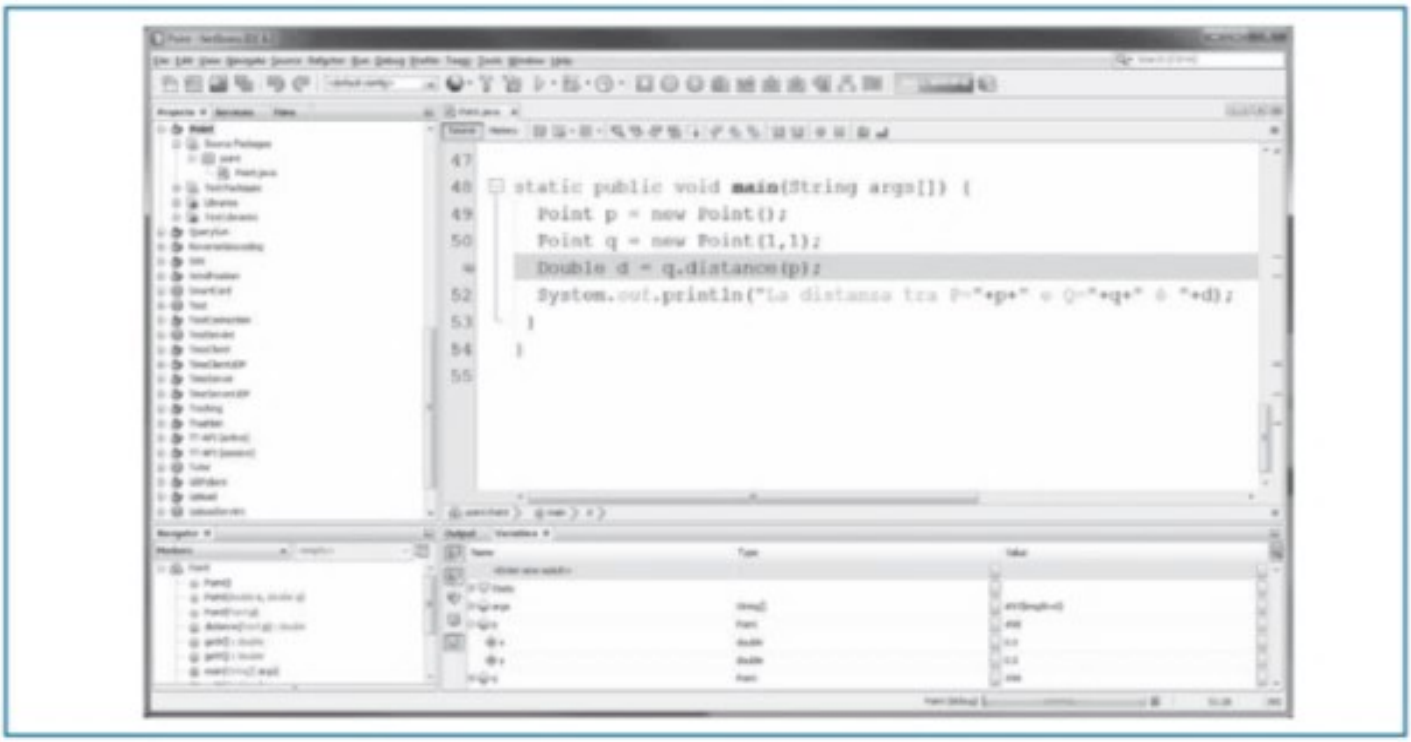
```
public class Point {
   private double x;
```


Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

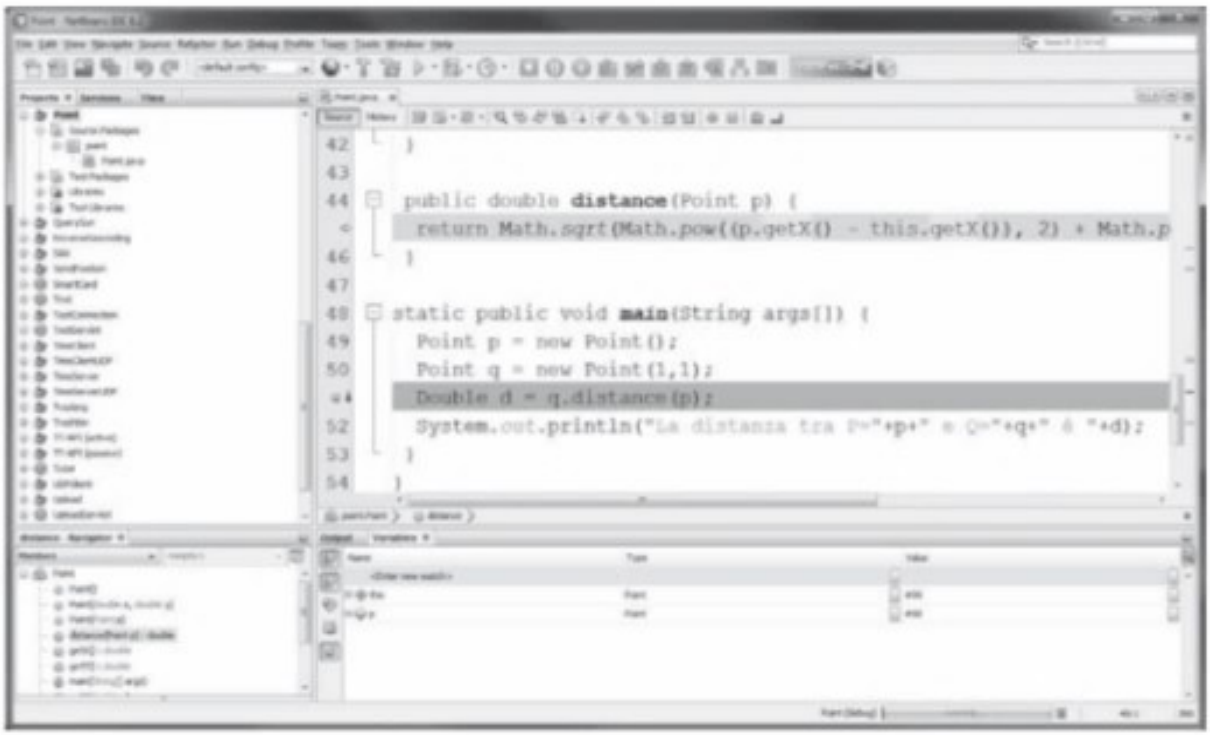
```
    private double y;

    public Point() {
      this.x = 0;
      this.y = 0;
    }

    public Point(double x, double y) {
      this.x = x;
      this.y = y;
    }

    public Point (Point p) {
      this.x = p.getX();
      this.y = p.getY();
    }

    public double getX() {
      return x;
    }

    public void setX(double x) {
      this.x = x;
    }

    public double getY() {
      return y;
    }

    public void setY(double y) {
      this.y = y;
    }

    public String toString() {
      String s = "("+this.getX()+";"+this.getY()+")";
      return s;
    }

    public double distance(Point p) {
      return Math.sqrt(Math.pow((p.getX() - this.getX()),2)+
             Math.pow((p.getY() - this.getY()),2));
    }

    static public void main(String args[]) {
      Point p = new Point();
      Point q = new Point(1,1);
      Double d = q.distance(p);
      System.out.println("La distanza tra P="+p+" e Q="+q+" è "+d);
    }
  }
```

L'impostazione di un punto di interruzione sulla riga di codice del metodo *main* che invoca il metodo *distance* sull'oggetto *q* di classe *Point* fornendo come argomento l'oggetto *p* consente di arrestare temporaneamente l'esecuzione del programma; la finestra di servizio *Variabili* permette in questa condizione di ispezionare in modo estremamente pratico e potente lo stato e il contenuto degli oggetti istanziati e dei loro attributi, fino a quelli di tipo primitivo.

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018



Una volta che un programma ha arrestato la sua esecuzione in corrispondenza di un punto di interruzione essa può essere ripresa con il comando *Continue* del menu *Debug*, oppure attivando l'icona dello specifico strumento. Ma se il codice presenta problemi che devono essere indagati è preferibile eseguire il programma «passo a passo» verificando i cambiamenti di valore delle variabili; a questo scopo è possibile utilizzare i comandi *Step Over* e *Step Into* del menu *Debug*: questi comandi sono di così comune utilizzo da parte del programmatore da essere associati rispettivamente ai tasti funzione *F8* ed *F7* (in alternativa è possibile attivare con il mouse le icone degli strumenti corrispondenti). Nel primo caso l'esecuzione prosegue alla riga di codice successiva, ma nel secondo se la riga di codice prevede l'invocazione di un metodo si «entra» nel codice dello stesso:



Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

2 Sviluppo di applicazioni con GUI utilizzando *Scene Builder*

NetBeans integra lo strumento *Scene Builder* per la realizzazione in modalità WYSIWYG (*What You See Is What You Get*) di applicazioni JavaFX dotate di un'interfaccia grafica per l'utente (GUI).

ESEMPIO La seguente classe *EasterCalendar* estende la classe *GregorianCalendar* della libreria Java impostando la data della Pasqua (calcolata con l'algoritmo di determinazione del «numero pasquale» del matematico e astronomo gesuita tedesco Clavius vissuto alla fine del XVI secolo[3]) dell'anno fornito come argomento del costruttore, o dell'anno in corso.

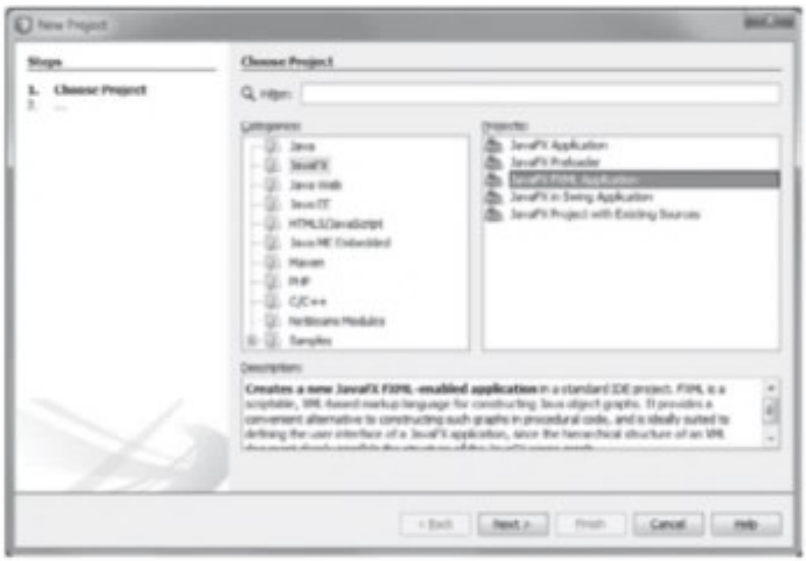
```
package easter;

import java.util.*;

public class EasterCalendar extends GregorianCalendar {

  private int easterNumber() {
    int y, g, c, x, z, d, e, n;

    y = this.get(Calendar.YEAR);
    g = (y%19) + 1;
    c = (y/100) + 1;
    x = 3*c/4 - 12;
    z = ((8*c + 5)/25) - 5;
    d = (5*y/4) - (x+10);
    e = (11*g + 20 + z - x)%30;
    if (((e == 25) && (g > 11)) || (e == 24))
      e++;
    n = 44 - e;
    if (n < 21)
      n += 30;
    n = n + 7 - ((d + n)%7);
    return n;
  }

  private void setEaster() {
    int easter_number = easterNumber();

    if (easter_number > 31) {
      int date = easter_number - 31;
      this.set(this.get(Calendar.YEAR),Calendar.APRIL,
               date,0,0,0);
    }
    else {
      int date = easter_number;
      this.set(this.get(Calendar.YEAR),Calendar.MARCH,
               date,0,0,0);
    }
  }

  public EasterCalendar() {
    super();
    setEaster();
  }
```

3 D. Knuth, *The Art of Computer Programming – Volume 1: Fundamental Algorithms*, Addison-Wesley, 1973.

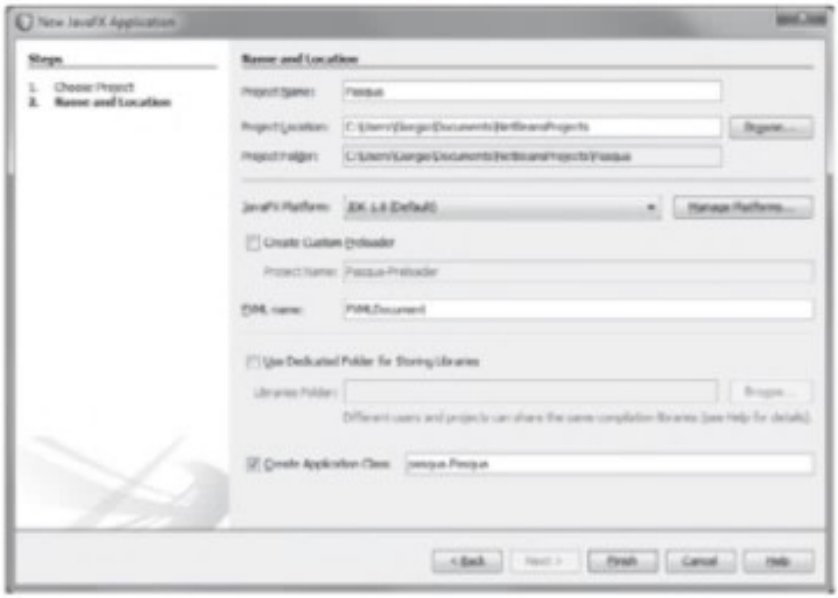
Questo file costituisce un'estensione online dei corsi di Fornichi, Meini, Vegnuti © Zanichelli editore 2018

```
  public EasterCalendar(int year) {
    super();
    if (year > 1582)
      this.set(Calendar.YEAR,year);
  setEaster();
  }
}
```

Per realizzare una semplice applicazione con interfaccia grafica da usarsi per la determinazione della data della Pasqua di un anno inserito dall'utente è necessario creare un progetto di tipo «JavaFX FXML Application» della categoria JavaFX:



Il *wizard* di creazione del progetto richiede il nome dello stesso e quello del file FXML che conterrà la descrizione della GUI dell'applicazione:



Il *wizard* crea il progetto generando automaticamente una bozza di file FXML, una classe *controller* e una *application* che non è normalmente necessario modificare.
Il menu contestuale che si apre agendo con il tasto destro del mouse sull'icona del file FXML consente di aprirlo in due diverse modalità: il comando *Edit* ne visualizza il contenuto in formato XML ▶

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

I CONCETTI CHIAVE

IDE *NETBEANS*. *NetBeans* è un IDE (*Integrated Development Environment*), cioè un ambiente integrato di codifica, compilazione e *debug* con interfaccia utente di tipo grafico e visuale per programmare in linguaggio Java.

PROGETTI DI SVILUPPO. Il concetto fondamentale nel contesto di un IDE è quello di «progetto»: un progetto è un insieme strutturato di file sorgente la cui compilazione produce come risultato il *byte-code* eseguibile di un programma Java.

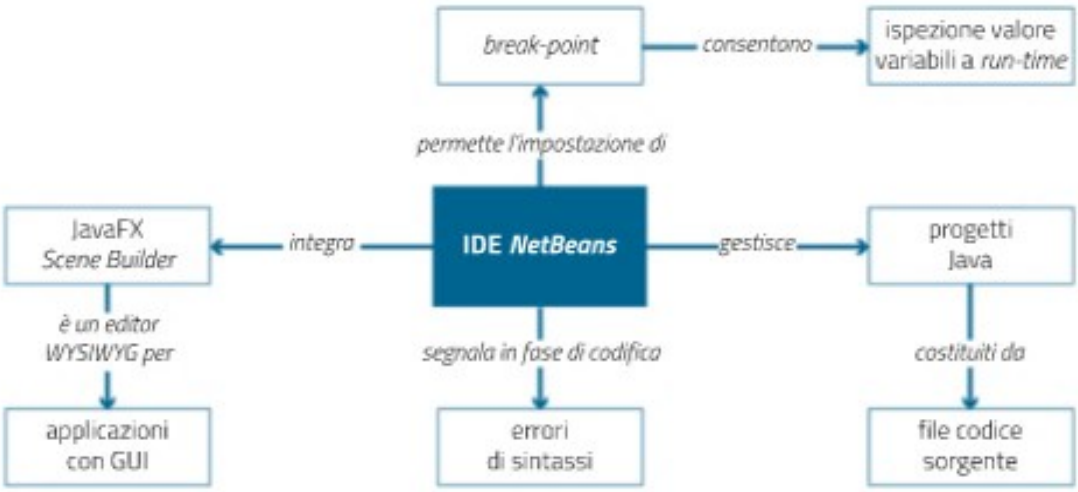
ERRORI DI SINTASSI. L'IDE *NetBeans* segnala gli errori di codifica nel corso della digitazione del codice da parte del programmatore.

***DEBUG* DI UN PROGRAMMA**. In *NetBeans* è possibile impostare «punti di interruzione» (*break-point*) in cui fermare l'esecuzione del codice per controllare il valore delle variabili allo scopo di effettuare il *debug* di un programma.

CREAZIONE VISUALE DI GUI. *NetBeans* integra lo strumento *Scene Builder* per la realizzazione in modalità WYSIWYG (*What You See Is What You Get*) di applicazioni JavaFX dotate di un'interfaccia grafica per l'utente (GUI).

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

RIPASSA CON LA MAPPA



Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

SEZIONE

B

Programmazione web lato client in linguaggio JavaScript

B1

La programmazione orientata agli oggetti in linguaggio JavaScript

JavaScript è un linguaggio debolmente tipizzato che nasce per rendere «dinamiche» le pagine HTML visualizzate dai browser. Le sue caratteristiche di flessibilità, nonché la possibilità di utilizzare diversi paradigmi di programmazione (imperativo, a oggetti o funzionale), fanno sì che sempre più sviluppatori lo scelgano come linguaggio di programmazione principale e in contesti sempre più diversificati: oltre che per realizzare applicazioni web lato client, sta riscuotendo successo anche per scrivere programmi web lato server, oppure per creare APP utilizzabili sui diversi sistemi operativi dei dispositivi *mobile*.

In questo capitolo verrà approfondito l'uso di JavaScript utilizzando il paradigma di programmazione a oggetti.

1 Programmazione a oggetti in JavaScript

In JavaScript gli oggetti possono essere di tre tipi:

- nativi (*built-in*);
- forniti dallo *host* (nel caso specifico dal browser);
- definiti dagli utenti.

Gli oggetti nativi sono completamente definiti dallo standard, indipendentemente dall'ambiente di esecuzione, come per esempio l'oggetto *Date* che rappresenta le date, o le collezioni *Map* e *Set* che vedremo successivamente. Quelli forniti dallo *host* dipendono dal contesto in cui il codice JavaScript viene eseguito: se in un browser web, lo *host* fornirà oggetti che permettono di interagire con la pagina e il browser stesso. Infine, quelli definiti dagli utenti, sono oggetti che ciascun programmatore costruisce per meglio modellare la soluzione al problema che deve risolvere.

In JavaScript ci sono diversi modi per definire un oggetto. In particolare, un oggetto può definire degli **attributi**, noti anche come **proprietà**, che sono variabili contenute nell'oggetto stesso.
L'insieme dei valori degli attributi o delle proprietà definisce lo stato che possiede un oggetto. La definizione può avvenire specificando, tra i simboli «{» e «}», coppie *variabile : valore* separate dal simbolo «,».
Il riferimento a una proprietà di un oggetto avviene congiungendo il nome dell'oggetto e il nome della proprietà separati dal simbolo «.».

I linguaggi tipizzati

Un linguaggio *debolmente tipizzato* è un linguaggio dove ogni dato ha associato un tipo. Lo è «debolmente» perché il tipo del dato può cambiare nel corso dell'esecuzione del codice. Altri linguaggi, come C++ e Java, si dicono *fortemente tipizzati* perché, oltre ad avere i tipi associati ai dati, i tipi dei dati non possono cambiare in fase di esecuzione. La tipizzazione debole ha alcuni vantaggi (principalmente di flessibilità), ma ha il limite che alcuni controlli sui tipi non sono possibili in fase di compilazione: in caso di un errore di tipo si ha un errore di esecuzione. Quando il controllo di tipo è utile, è possibile far riferimento a una variante di JavaScript, **TypeScript**, dove è possibile associare i tipi ai dati e verificarli durante la compilazione. TypeScript è un sovra-insieme di JavaScript; tra i suoi punti di forza bisogna segnalare che non rinuncia alla compatibilità con lo standard perché la compilazione di un programma TypeScript produce un programma JavaScript.

ESEMPIO

```
var persona = {
  nome : "Ivan Venuti",
  natoIl : new Date(1972, 2, 29),
};
console.log(persona.nome);
console.log(persona.natoIl);
```

L'invocazione della funzione *console.log* (dove *console* è un oggetto reso disponibile dallo *host*) visualizza – per scopi esclusivamente di *debug* del codice – la variabile fornita come argomento; tutti i browser rendono disponibile, tra gli strumenti per gli sviluppatori, l'accesso alla finestra di visualizzazione.

Oltre alle proprietà, è possibile definire dei **metodi**, funzioni definite all'interno dell'oggetto che ne modellano il comportamento. Il riferimento a un metodo di un oggetto avviene congiungendo il nome dell'oggetto e il nome del metodo con il simbolo «.».
La parola chiave `this` permette di riferire una proprietà o un metodo dell'oggetto in cui è definito il codice che la utilizza.

ESEMPIO

```
var persona = {
  nome : "Ivan Venuti",
  natoIl : new Date(1972, 2, 29),

  età() {
    let diff = Date.now() - this.natoIl.getTime();
    return (new Date(diff)).getUTCFullYear() - 1970;
  }

};
console.log(persona.età());
```

JavaScript permette di definire dinamicamente i nomi delle proprietà o dei metodi, cioè permette di usare il risultato di calcoli per creare il nome al momento della definizione.

ESEMPIO

```
var indice=1;
var oggetto1 = {
  ["prop" + indice++] : "Una proprietà",
  ["prop" + indice++] : indice,
};
var oggetto2 = {
  ["prop" + indice++] : "Altra proprietà",
  ["prop" + indice++] : indice,
  ["incrementa" + indice](a) { return a + 1; }
};
console.log(oggetto1.prop1); // visualizza: "Una proprietà"
console.log(oggetto1.prop2); // visualizza: "3"
console.log(oggetto2.prop3); // visualizza: "Altra proprietà"
console.log(oggetto2.prop4); // visualizza: "5"
console.log(oggetto2.incrementa5(3)); // visualizza: "4"
```

Linguaggi orientati agli oggetti

Il paradigma a oggetti rappresenta indubbiamente un'astrazione molto vicina al modo di pensare e rappresentare la realtà delle persone. Per questo motivo è un paradigma che ha avuto, e sta avendo, un enorme successo. Il linguaggio C++ nasce con l'intento di estendere il C con i costrutti tipici della programmazione orientata agli oggetti. Java nasce da subito come linguaggio a oggetti. Il linguaggio PHP ha introdotto il concetto di classi a partire dalla versione 5. Sorprendentemente JavaScript introduce il concetto di classe solo a partire dalla versione ECMAScript 6. Nonostante questo, ancora oggi il linguaggio si basa su un concetto diverso dalla classe e che fa uso di riferimenti ad altri oggetti, realizzando un meccanismo di «delega» al posto del concetto di estensione tipico del paradigma *object-oriented*.

1.1 Classi e oggetti

Anziché definire un singolo oggetto, è possibile, a partire da ECMAScript 6, utilizzare il concetto di classe, ovvero definire la struttura generale che ogni oggetto di quella classe avrà.

Una **classe**, nella terminologia a oggetti, permette di definire un insieme di proprietà e un insieme di metodi comuni a diversi oggetti. Lo speciale metodo `constructor` definisce e inizializza le proprietà dell'oggetto che verrà creato a partire dalla classe utilizzando la parola chiave `new`.

ESEMPIO

Il codice che segue modella un segmento a due dimensioni utilizzando quattro coordinate (*x* e *y* del punto d'inizio e *x* e *y* del punto di fine del segmento):

```
class Segmento {
   constructor(x1, y1, x2, y2) {
      this.x1 = x1;
      this.y1 = y1;
      this.x2 = x2;
      this.y2 = y2;
   }

   disegna(canvas) {
      canvas.beginPath();
      canvas.moveTo(this.x1, this.y1);
      canvas.lineTo(this.x2, this.y2);
      canvas.stroke();
   }

   toString() {
      return "Segmento: x1=" + this.x1 + " ; y1=" + this.y1 + " ; x2=" + this.x2 + " ;
      y2=" + this.y2;
   }
}

var s = new Segmento(1, 2, 3, 4);
console.log("segmento: " + s);
```

Il metodo *toString* di un oggetto viene invocato automaticamente quando è necessario trasformare l'oggetto in una stringa di caratteri, come nel caso della visualizzazione di *debug* dell'esempio precedente.

È comune che una stringa di *debug* sia generata concatenando il valore di più variabili, ciascuna preceduta da una stringa di descrizione. In questo caso l'istruzione diventa ben poco leggibile e fonte di errori di digitazione ed è opportuno ricorrere alle *template string*.

Una *template string* è una stringa delimitata dal simbolo «`» (*backtick*). Al suo interno è possibile scrivere il valore di una variabile, o di un'espressione, utilizzando la sintassi: *${espressione}*. Una *template string* può essere scritta anche su più righe.

Oltre al valore di una variabile è possibile specificare una qualsiasi istruzione la cui esecuzione avviene al momento della visualizzazione. Per esempio l'istruzione `console.log(`${x+2}`);` visualizzerà il valore della variabile *x* aumentato di 2.

ESEMPIO

```
class Segmento {
  ...
  ...
  ...
  toString() {
    return
      `Segmento:
        x1=${this.x1};
        y1=${this.y1};
        x2=${this.x2};
        y2=${this.y2}`;
  }
}
```

1.2 Ereditarietà di classe

È possibile definire *classi derivate* con la parola chiave **extends**: esse ereditano dalla classe tutte le caratteristiche (proprietà e metodi) e possono definirne di nuove; è inoltre possibile ridefinire i metodi o le proprietà. La ridefinizione «maschera» il metodo o la proprietà della classe da cui deriva: la parola chiave **super** permette di riferirsi a proprietà e metodi della classe originale.

ESEMPIO

```
class SegmentoColorato extends Segmento {
  constructor(x1, y1, x2, y2, colore) {
    super(x1, y1, x2, y2);
    this.colore = colore;
  }
  disegna(canvas) {
    canvas.beginPath();
    canvas.strokeStyle = this.colore;
    super.disegna(canvas);
    canvas.stroke();
  }
  toString() {
    return super.toString() + ' colore=' + this.colore;
  }
}
```

ESEMPIO

Nella pagina HTML che segue sono utilizzate le classi *Segmento* e *SegmentoColorato* per disegnare su un *canvas* HTML un segmento ogni volta che si muove il mouse (evento *mousemove*) e cambiarne il colore a ogni click (evento *click*):

```
<!doctype html>
<html>
  <head>
    <script>
      'use strict';
      // definizioni delle classi Segmento e SegmentoColorato
```

```
        ...
        ...
        ...
        var colori = ["red", "green", "magenta", "purple", "yellow", "black"];
        var linea;
        var contesto;

        function disegna() {
          var tag = document.getElementById("mioCanvas");
          contesto = tag.getContext("2d");
          linea = new SegmentoColorato(
            200, 200, 200, 200,
            colori[
              Math.floor(Math.random() * colori.length)
            ]);
        }

        function mousemove(evt) {
          if (linea) {
            linea.x1 = linea.x2;
            linea.y1 = linea.y2;
            linea.x2 = evt.clientX;
            linea.y2 = evt.clientY;
            linea.disegna(contesto);
          }
        }

        function cambiacolore() {
          if (linea)
            linea.colore = colori[Math.floor(Math.random() * colori.length)];
        }

        document.onmousemove = mousemove;
        document.onclick = cambiacolore;
        </script>
      </head>
      <body onload="disegna()">
        <canvas width="400" height="400" id="mioCanvas" style="border:1px solid #000000;">
        </canvas>
    </body>
  </html>
```

1.3 Ereditarietà prototipale

In JavaScript ogni oggetto possiede una particolare proprietà denominata *prototype* (**prototipo**) che è un riferimento a un altro oggetto. Come suggerisce il nome, questa proprietà serve a fornire all'oggetto un «modello», cioè un oggetto di partenza a cui aggiungere altre proprietà o metodi. Qualsiasi proprietà o metodo aggiunto al prototipo viene automaticamente ereditato da tutti gli oggetti creati.

Quando si crea un oggetto utilizzando la parola chiave `new` seguita dal nome di una funzione si definisce la funzione stessa come prototipo dell'oggetto. Invocando `new` più volte con la stessa funzione si creano più oggetti aventi un prototipo comune.

ESEMPIO

```
function Libro(titolo, prezzo, editore) {
   this.titolo = titolo;
   this.prezzo = prezzo;
   this.editore = editore;
}

Libro.prototype.sconto = 10;
var mioLibro = new Libro('Javascript or ora', 40, 'Zanichelli');
var altroLibro = new Libro('Javascript sempre', 35, 'Zanichelli');
console.log(mioLibro.sconto);   // visualizza: "10"
console.log(altroLibro.sconto); // visualizza: "10"
```

Se la proprietà o il metodo non sono definiti nell'oggetto, devono essere ricercati nel prototipo: se non sono presenti nel prototipo, allora la proprietà o il metodo vengono ricercati risalendo i prototipi in catena fino a quando non li si trovano definiti, oppure il prototipo ha come valore della proprietà *prototype* il valore `null`. Al termine della catena dei prototipi c'è sempre *Object.prototype* (a cui, per esempio, appartiene il metodo *toString*). Se sono presenti più proprietà o metodi, nell'oggetto e nel prototipo, o nella catena dei prototipi, ha precedenza quello incontrato per primo.

ESEMPIO

```
function Libro(titolo, prezzo, editore) {
   this.titolo = titolo;
   this.prezzo = prezzo;
   this.editore = editore;
}

Libro.prototype.sconto = 10;
var mioLibro = new Libro('Javascript or ora', 40, 'Zanichelli');
mioLibro.sconto = 20;
   console.log(mioLibro.sconto); // visualizza: "20"
```

L'istruzione *mioLibro.sconto = 20;* crea una nuova proprietà *sconto* nell'oggetto *mioLibro* che maschera la proprietà *sconto* del relativo prototipo.
Ma *Libro* è una funzione, che significato può avere l'espressione *Libro.prototype.sconto*? *Libro* è anche un oggetto perché tutte le funzioni JavaScript sono, a loro volta, oggetti aventi la caratteristica di risultare invocabili: a questo, come a tutti gli oggetti, è possibile aggiungere nuove proprietà.

Creare oggetti invocando *Object.create*

La parola chiave `new` seguita dal nome di una funzione «costruttore» crea un oggetto con le proprietà definite all'interno della funzione; inoltre il suo prototipo viene collegato – copiandone il riferimento – al prototipo della funzione. Gli oggetti istanza di una classe sono in realtà oggetti aventi il prototipo in comune.

È possibile specificare il prototipo di un oggetto in fase di creazione. Infatti, oltre a poter creare un oggetto ricorrendo alla parola chiave `new` o enumerando le proprietà e i metodi, esiste un terzo modo per creare un oggetto JavaScript: invocare la funzione *Object.create* che ha come parametro un oggetto da usare come prototipo.

ESEMPIO

```
var mioLibro = {
  titolo: 'Javascript: una meraviglia',
  editore: 'Zanichelli'
};

var mioLibroEbook = Object.create(mioLibro);
mioLibroEbook.formato = "digitale";
// visualizza: "Javascript: una meraviglia"
console.log(mioLibroEbook.titolo);
```

OSSERVAZIONE Dopo aver chiarito il vero significato dei costruttori, è lecito chiedersi se le parole chiave introdotte nel linguaggio per la gestione delle classi (`class`) e l'ereditarietà (`extends`) introducano delle significative differenze rispetto ai prototipi e alla catena dei prototipi. La risposta è no: tali parole chiave non sono altro che degli artifici sintattici. Rappresentano comunque un importante ausilio per chi si avvicina a JavaScript avendo una conoscenza di un diverso linguaggio di programmazione orientato agli oggetti.

1.4 Estendere i tipi predefiniti

L'utilizzo del prototipo è un meccanismo potente e flessibile, tanto che può essere utilizzato per ridefinire i tipi di dato predefiniti del linguaggio.

ESEMPIO

Ecco come estendere l'oggetto *String* affinché esponga un nuovo metodo denominato *palindroma* che restituisce `true` se la stringa è palindroma (come nel caso di «etnagigante»), `false` altrimenti:

```
String.prototype.palindroma = function() {
  for (var i=0; i<this.length / 2; i++)
    if (this.charAt(i)!=this.charAt(this.length-i-1))
      return false;
  return true;
};
console.log("etnagigante".palindroma()); // visualizza: "true"
```

OSSERVAZIONE Ridefinire il comportamento di un oggetto predefinito è un'operazione rischiosa, soprattutto se ha un impatto su funzionalità esistenti; infatti potrebbe accadere che dei programmi o librerie si basino sul comportamento standard per svolgere le proprie funzionalità.

2 Collezioni di oggetti

In ECMAScript 6 sono state introdotte quattro nuove strutture dati: *Map*, *WeakMap*, *Set* e *WeakSet*. Queste strutture dati sono collezioni non ordinate di oggetti. Le due di tipo *Weak* sono strutture che permettono al *garbage collector* di recuperare gli oggetti in esse presenti qualora gli oggetti restino referenziati unicamente all'interno delle strutture.

2.1 Mappe

Le **mappe** (*Map*) permettono di memorizzare coppie *chiave/valore*, dove sia la chiave sia il valore possono essere valori o oggetti qualsiasi.

Gli oggetti JavaScript non possono essere usati per questo scopo usando il nome delle proprietà come chiave, perché i nomi delle proprietà di un oggetto possono essere solo di tipo stringa, vincolo non presente nelle mappe.

ESEMPIO Il codice che segue illustra l'uso di *Map* per memorizzare coppie chiave/valore di oggetti confrontandolo con l'uso di un oggetto per lo stesso scopo:

```
var map = new Map();
map.set("3470000", "Telefono1");
map.set(3470000, "Telefono2");
console.log(map.get(3470000));     // visualizza: "Telefono2"
console.log(map.get("3470000"));   // visualizza: "Telefono1"

var obj = new Object();
obj["3470000"] = "Telefono1";
obj[3470000] = "Telefono2";
console.log(obj[3470000]);         // visualizza: "Telefono2"
console.log(obj["3470000"]);       // visualizza: "Telefono2"
```

OSSERVAZIONE Quando si usano coppie chiave/valore dove la chiave è una stringa, è possibile scegliere se usare *Map* oppure un oggetto, dove si impiegano le proprietà come chiavi a cui associare un valore. Quando le chiavi non sono stringhe è necessario usare le mappe. In ogni caso le chiavi devono essere comparabili per valore (gli oggetti, per esempio, sono considerati uguali non se hanno lo stesso contenuto, ma se hanno la stessa identità).

La tabella che segue riepiloga i metodi e le proprietà più usate per la gestione di una collezione di oggetti di tipo *Map*.

Metodo o proprietà	Descrizione
`clear()`	Elimina tutti gli elementi.
`delete(chiave)`	Elimina dalla mappa l'elemento con la chiave specificata.
`entries()`	Restituisce un oggetto iterabile dove ogni elemento è un *array* [*chiave, valore*]; l'ordine degli elementi è quello con cui sono stati inseriti.
`forEach(callback)`	Invoca la funzione *callback* per ogni elemento della mappa.
`get(chiave)`	Restituisce il valore associato alla chiave specificata; restituisce **undefined** se non esiste un elemento con la chiave specificata.
`has(chiave)`	Restituisce un valore di verità che indica se nella mappa c'è o meno l'elemento con la chiave specificata.
`keys()`	Restituisce un oggetto iterabile contenente le chiavi presenti nella mappa (l'ordine è quello di inserimento degli elementi).
`Map([iterabile])`	È un costruttore che permette di creare una mappa partendo da un qualsiasi oggetto iterabile, i cui elementi devono essere *array* di tipo [*chiave, valore*].
`set(chiave, valore)`	Aggiunge un elemento associando il valore alla chiave specificata.
`size`	È la proprietà che contiene il numero di elementi della mappa.
`values()`	Restituisce un oggetto iterabile contenente i valori presenti nella mappa (l'ordine è quello di inserimento degli elementi).

Garbage collector

JavaScript, come altri linguaggi di programmazione, permette al programmatore di specificare quando deve essere allocata una variabile, ma gestisce in modo automatico il suo rilascio. L'algoritmo che lavora dietro le quinte e colleziona le risorse che possono essere rilasciate perché «non servono più» è denominato *garbage collector*. Esistono vari modi per realizzarlo, quello dell'interprete JavaScript si basa sul conteggio dei riferimenti, ovvero sul numero di oggetti o contesti che si riferiscono all'oggetto: quando tale numero è 0, il *garbage collector* rilascia lo spazio di memoria utilizzato. È un algoritmo che presenta alcuni difetti, per esempio nel caso di riferimenti circolari per i quali il *garbage collector* non è in grado di recuperare gli oggetti che vi appartengono. Un algoritmo noto che risolve questo problema è denominato *mark-and-sweep*: partendo da oggetti «radice» si marcano tutti gli oggetti da essi raggiungibili: dopo la marcatura gli oggetti non marcati possono essere liberati; questo algoritmo è purtroppo più lento del precedente. Il programmatore può ignorare quale algoritmo viene utilizzato, ponendosi solo il problema di non lasciare inutili riferimenti a oggetti usando, per esempio, le strutture di tipo *Weak* quando è opportuno.

2.2 Insiemi

Gli **insiemi** di oggetti sono realizzati con l'oggetto *Set*; essi sono collezioni di valori univoci (cioè non possono comparire più di una volta). È possibile iterare gli elementi di un insieme: in questo caso viene preservato l'ordine di inserimento.

ESEMPIO

```
var insieme = new Set("Nel mezzo del cammin di nostra vita");

console.log("insieme: " + [...insieme]);
console.log(insieme.has("n"));
console.log(insieme.has("h"));
/*
visualizza:
  "insieme: N,e,l, ,m,z,o,d,c,a,i,n,s,t,r,v"
  "true"
  "false"
*/
```

La tabella che segue riepiloga i metodi e le proprietà più usate per la gestione di una collezione di oggetti di tipo *Set*.

Metodo o proprietà	Descrizione
`add(valore)`	Aggiunge un valore all'insieme.
`clear()`	Elimina tutti gli elementi dall'insieme.
`delete(valore)`	Elimina dall'insieme l'elemento con il valore specificato.
`entries()`	Restituisce un oggetto iterabile dove ogni elemento è un *array* [*valore*, *valore*], in modo da rendere il risultato simile al metodo omonimo della collezione *Map*; l'ordine degli elementi è quello con cui sono stati inseriti.
`forEach(callback)`	Chiama la funzione *callback* per ogni elemento dell'insieme.
`has(valore)`	Restituisce un valore di verità che indica se l'elemento appartiene o meno all'insieme.
`keys()`	Fornisce lo stesso risultato del metodo *values*.
`Set([iterabile])`	È un costruttore che permette di creare un insieme partendo da un qualsiasi oggetto iterabile; se non c'è il parametro (o se il parametro è **`null`**), l'insieme è inizialmente vuoto.
`size`	È la proprietà che contiene il numero di elementi dell'insieme.
`values()`	Restituisce un oggetto iterabile contenente i valori presenti nell'insieme (l'ordine è quello di inserimento degli elementi).

2.3 Operazioni su mappe e insiemi

In JavaScript i vettori sono una struttura molto flessibile e usata in vari contesti.

Per trasformare un oggetto di tipo *Map* o *Set* in un vettore si può ricorrere all'operatore *spread*: [...]. Tale operatore si applica a un qualsiasi oggetto iterabile e lo espande nei suoi elementi: è utile sia nel caso di argomenti di funzione sia, come nell'esempio che segue, per trasformare le chiavi e i valori di una mappa in elementi di un vettore.

ESEMPIO

```
var map = new Map();
map.set("3470000", "Telefono1");
map.set(3470000, "Telefono2");
var tutti = [...map.keys(), ...map.values()];
for (var valore in tutti)
  console.log(typeof(tutti[valore]) + ": " + tutti[valore]);
/*
  visualizza:
  "string: 3470000"
  "number: 3470000"
  "string: Telefono1"
  "string: Telefono2"
*/
```

OSSERVAZIONE L'operatore `typeof` impiegato nell'esempio precedente permette di ottenere il tipo del valore fornito come argomento, nel caso specifico per visualizzarlo.

3 Moduli per applicazioni complesse

La realizzazione di applicazioni complesse comporta l'utilizzo di numerosi oggetti e/o funzioni. Anziché replicare parti di codice simili in file HTML distinti, è preferibile includere uno o più file esterni. In questo modo si ottengono i seguenti vantaggi:

- il browser può riconoscere che i file esterni inclusi sono gli stessi delle altre pagine e decidere di leggerli dalla *cache* locale anziché ricaricarli dal server: questo porta a un vantaggio in termini di minor tempo di download della pagina;
- eventuali modifiche e/o miglioramenti sono subito disponibili per tutte le pagine, agevolando il compito di test e manutenzione;
- una netta separazione tra la parte di contenuto (HTML), di presentazione (CSS) e di comportamento (JavaScript) agevola la comprensione delle singole parti, semplifica il processo di sviluppo e offre la possibilità di una suddivisione di compiti all'interno dei gruppi di lavoro;
- una netta separazione tra la parte di contenuto, di presentazione e di comportamento semplifica anche la validazione delle pagine: i file HTML e i fogli CSS possono essere validati separatamente dal codice JavaScript.

OSSERVAZIONE È sempre bene valutare il ruolo che gli *script* avranno nella propria applicazione e fare in modo di dare funzionalità equivalenti ai browser che non interpretano il codice JavaScript (o perché obsoleti, o dotati di interfacce non visuali per persone con disabilità o, semplicemente, perché l'utente ha disabilitato l'uso degli *script*). Un modo per farlo è ricorrere al *tag* `<noscript>` che viene usato solo quando l'esecuzione di uno *script* non è possibile.

3.1 Suddivisione in moduli

A partire da ECMAScript 6 è possibile creare dei moduli per inserire funzionalità logicamente correlate e includerli nelle proprie applicazioni. Un **modulo** è un file JavaScript esterno in cui ci sono frammenti di codice (funzioni, classi o semplici variabili) che sono esportate utilizzando la parola chiave `export`.

ESEMPIO

```
// file lib/colori.js
export var colorePreferito = "#885533";

export function coloreCasuale() {
  return '#' + casualeBase16()
  +casualeBase16()
  +casualeBase16()
  +casualeBase16()
  +casualeBase16()
  +casualeBase16();
}

function casualeBase16() {
  return Math.floor(Math.random()*16).toString(16);
}
```

Per poter utilizzare un modulo è necessario specificare l'attributo *type* del *tag* `<script>`: il valore deve essere *module*. Inoltre è necessaria la direttiva `import` seguita dalla tipologia dell'importazione (lista di nomi di variabili o funzioni, separate dal simbolo «,» racchiusa tra i simboli «{» e «}), oppure utilizzando il simbolo «*» per includere tutto quanto contenuto nel file; in quest'ultimo caso deve essere indicato un *alias* da usare per accedere ai nomi importati.

ESEMPIO

```
<script type="module">
  import * as col from "./lib/colori.js"
  import {bello, meraviglioso} from "./lib/coseOttime.js"

  console.log(col.colorePreferito);
  console.log(col.coloreCasuale());
  console.log(bello);
</script>
```

I CONCETTI CHIAVE

OGGETTI. In JavaScript gli oggetti possono essere di tre tipi: nativi (*built-in*), forniti dallo *host* (nel caso specifico dal browser) e definiti dagli utenti.

PROPRIETÀ. In JavaScript ci sono diversi modi per definire un oggetto. In particolare, un oggetto può definire degli *attributi*, noti anche come *proprietà*, che sono variabili contenute nell'oggetto stesso. L'insieme dei valori degli attributi o delle proprietà definisce lo stato che possiede un oggetto. La definizione può avvenire specificando, tra i simboli «{» e «}», coppie *variabile : valore* separate dal simbolo «,». Il riferimento a una proprietà di un oggetto avviene congiungendo il nome dell'oggetto e il nome della proprietà separati dal simbolo «.».

METODI. Oltre alle proprietà, è possibile definire dei metodi, funzioni definite all'interno dell'oggetto che modellano il comportamento degli oggetti. Il riferimento a un metodo di un oggetto avviene congiungendo il nome dell'oggetto e il nome del metodo separati dal simbolo «.».
La parola chiave `this` permette di riferire una proprietà o un metodo dell'oggetto in cui è definito il codice che la utilizza.

CLASSI. Una classe, nella terminologia a oggetti, permette di definire un insieme di proprietà e un insieme di metodi comuni a diversi oggetti. Lo speciale metodo `constructor` definisce e inizializza le proprietà dell'oggetto che verrà creato a partire dalla classe utilizzando la parola chiave `new`.

EREDITARIETÀ DI CLASSE. È possibile definire *classi derivate* con la parola chiave `extends`: esse ereditano dalla classe tutte le caratteristiche (proprietà e metodi) e possono definirne di nuove; è inoltre possibile ridefinire i metodi o le proprietà. La ridefinizione «maschera» il metodo o la proprietà della classe da cui deriva: la parola chiave `super` permette di riferirsi a qualcosa di specifico della classe originale.

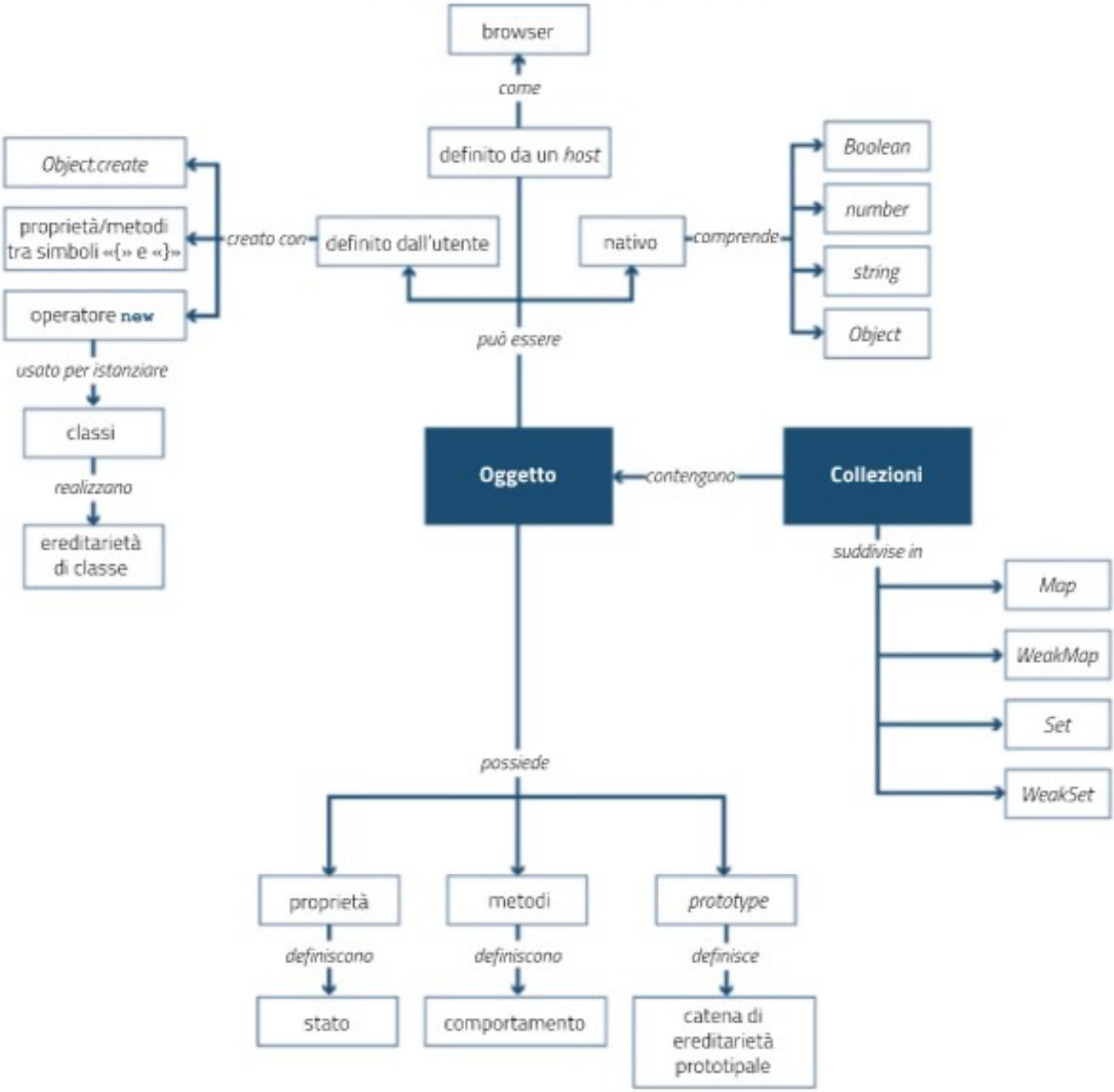
EREDITARIETÀ PROTOTIPALE. In JavaScript ogni oggetto possiede una particolare proprietà denominata *prototype* (*prototipo*) che è un riferimento a un altro oggetto. Come suggerisce il nome, questa proprietà serve a fornire all'oggetto un «modello», cioè un oggetto di partenza a cui aggiungere altre proprietà o metodi. Qualsiasi proprietà o metodo aggiunto al prototipo viene automaticamente ereditato da tutti gli oggetti creati.
Se la proprietà o il metodo non sono definiti nell'oggetto, devono essere ricercati nel prototipo: se non sono presenti nel prototipo, allora la proprietà o il metodo vengono ricercati risalendo i prototipi in catena fino a quando non li si trovano definiti, oppure il prototipo ha come valore della proprietà *prototype* il valore `null`.
Al termine della catena dei prototipi c'è sempre *Object.prototype* (a cui, per esempio, appartiene il metodo *toString*). Se sono presenti più proprietà o metodi, nell'oggetto e nel prototipo, o nella catena dei prototipi, ha precedenza quello incontrato per primo.

CREAZIONE DI OGGETTI. È possibile specificare il prototipo di un oggetto in fase di creazione. Infatti, oltre a poter creare un oggetto ricorrendo alla parola chiave `new` o enumerando le proprietà e i metodi, esiste un terzo modo per creare un oggetto JavaScript: invocare la funzione *Object.create* che ha come parametro un oggetto da usare come prototipo.

COLLEZIONI DI OGGETTI. In ECMAScript 6 sono state introdotte quattro nuove strutture dati: *Map*, *WeakMap*, *Set* e *WeakSet*. Queste strutture dati sono collezioni non ordinate di oggetti. Le due di tipo *Weak* sono strutture che permettono al *garbage collector* di recuperare gli oggetti in esse presenti qualora gli oggetti restino referenziati unicamente all'interno delle strutture.
Le *mappe* (*Map*) permettono di memorizzare coppie *chiave/valore*, dove sia la chiave sia il valore possono essere valori o oggetti qualsiasi.
Gli *insiemi* di oggetti sono realizzati con l'oggetto *Set*; essi sono collezioni di valori univoci (cioè non possono comparire più di una volta). È possibile iterare gli elementi di un insieme: in questo caso viene preservato l'ordine di inserimento.

MODULI. A partire da ECMAScript 6 è possibile creare dei moduli per inserire funzionalità logicamente correlate e includerli nelle proprie applicazioni.
Un modulo è un file JavaScript esterno in cui ci sono frammenti di codice (funzioni, classi o semplici variabili) che sono esportate utilizzando la parola chiave `export`.

RIPASSA CON LA MAPPA



SPIEGA IN UNA PAGINA

1 **Descrivere, anche mediante esempi di codice, com'è possibile definire e creare oggetti utilizzando il linguaggio JavaScript.**

2 **Illustrare le principali tipologie di collezioni presenti nel linguaggio JavaScript e le loro differenze.**

QUESITI A RISPOSTA APERTA

3 **In quali contesti si usano le parole chiave `this` e `super` nel codice del metodo di una classe?**

4 **È possibile aggiungere metodi agli oggetti predefiniti del linguaggio JavaScript?**

5 **Confrontare i metodi *keys* e *values* di un oggetto di tipo *Map* e uno di tipo *Set*.**

TEST A RISPOSTA MULTIPLA

6 **Un linguaggio è debolmente tipizzato quando:**

A non ha i tipi associati ai dati.
B ha i tipi associati ai dati, ma tali tipi possono cambiare.
C ha i tipi associati ai dati e tali tipi non possono cambiare.
D non presenta il concetto di tipo.

7 **In JavaScript quali tra i seguenti tipi di oggetto non sono presenti?**

A nativi (o *built-in*).
B specifici.
C forniti dallo *host*.
D definiti dagli utenti.

8 **Il metodo `constructor` serve a:**

A verificare che ci sia memoria disponibile per creare l'oggetto.
B specificare una classe da cui ereditare proprietà e metodi.
C assegnare un valore iniziale alle proprietà dell'oggetto quando viene creato.
D liberare la memoria allocata a un oggetto.

9 **La parola chiave `extends`:**

A permette di derivare una nuova classe a partire da una classe esistente.
B permette di estendere il range del tipo numerico predefinito del linguaggio.
C permette di estendere la dimensione di una collezione di oggetti.
D permette di definire un nuovo tipo di dati.

10 **Il prototipo di un oggetto (*prototype*):**

A riferisce un oggetto in cui ricercare proprietà e metodi eventualmente non presenti nell'oggetto stesso.
B riferisce una funzione speciale che consente di inizializzare un oggetto al momento della creazione.
C riferisce una proprietà speciale che contiene il nome dell'oggetto stesso.
D Gli oggetti del linguaggio JavaScript non hanno un prototipo.

11 **Nel caso che una classe abbia ridefinito le proprietà *nome* della classe da cui deriva, come si accede alla proprietà della classe originale?**

A `this.nome`.
B `super.nome`.
C `padre.nome`.
D `extends.nome`.

12 **Creare un Oggetto *Set* passando al costruttore una stringa di caratteri ha l'effetto di creare un insieme con:**

A un elemento.
B tanti elementi quanti sono i caratteri della stringa.
C tanti elementi quanti sono i caratteri diversi della stringa.
D associata una descrizione che è il contenuto della stringa.

13 **Per creare un modulo, è necessario creare un file al cui interno:**

A tutte le funzioni create sono utilizzabili dall'esterno.
B solo le funzioni precedute da `export` sono utilizzabili dall'esterno.
C solo le funzioni precedute da `public` sono utilizzabili dall'esterno.
D nessuna funzione è utilizzabile dall'esterno.

14 Per includere un modulo dentro un programma JavaScript, si utilizza il *tag*:

A `<script type= "module">`.

B `<module file= "...">`.

C `<module type= "...">`.

D `<script file= "...">`.

PROBLEMI

15 Definire una classe JavaScript che modelli un generico indirizzo comprensivo di via, numero civico, città e provincia.

16 Definire una classe JavaScript che modelli l'anagrafica di una persona comprensiva di nome, cognome, data di nascita, luogo di nascita, indirizzo di residenza e indirizzo di domicilio (per gli indirizzi usare la classe definita nel problema precedente).

17 Definire una classe JavaScript che modelli una persona che includa, oltre all'anagrafica, anche i contatti: telefono, indirizzo e-mail, sito web, ...).

18 Utilizzare un'opportuna collezione, o un insieme di collezioni, per creare uno *script* JavaScript che permetta di associare a diverse discipline (Scienze, Fisica, Matematica, ...) un insieme di *link* di materiali collegati.

CASI PRATICI

19 Creare e verificare una classe che permetta di impostare nome, cognome, data di nascita e sesso di una persona e fornisca un metodo *verificaCodiceFiscale* che restituisce `true` quando il codice fiscale è corretto rispetto alle informazioni fornite, `false` altrimenti (ricercare in rete la modalità con cui viene costruito il codice fiscale di una persona fisica).

20 Creare e testare un modulo che permetta di esporre le funzioni statistiche di media e varianza di un insieme di dati numerici.

to dredge
dragare

to inherit
ereditare

messy
disordinato

to nest
annidare

to retrieve
recuperare

to wrap
avvolgere

Objects 3

Upon a homely object Love can wink.
–William Shakespeare, *The Two Gentlemen of Verona*

The simple types of JavaScript are numbers, strings, Booleans (`true` and `false`), `null`, and `undefined`. All other values are *objects*. Numbers, strings, and Booleans are object-like in that they have methods, but they are immutable. Objects in JavaScript are mutable keyed collections. In JavaScript, arrays are objects, functions are objects, regular expressions are objects, and, of course, objects are objects.

An object is a container of properties, where a property has a name and a value. A property name can be any string, including the empty string. A property value can be any JavaScript value except for `undefined`.

Objects in JavaScript are class-free. There is no constraint on the names of new properties or on the values of properties. Objects are useful for collecting and organising data. Objects can contain other objects, so they can easily represent tree or graph structures.

JavaScript includes a prototype linkage feature that allows one object to inherit the properties of another. When used well, this can reduce object initialisation time and memory consumption.

Object Literals

Object literals provide a very convenient notation for creating new object values. An object literal is a pair of curly braces surrounding zero or more name/value pairs. An object literal can appear anywhere an expression can appear:

```
var empty_object = {};
var stooge = {
"first-name": "Jerome",
"last-name": "Howard"
};
```

A property's name can be any string, including the empty string. The quotes around a property's name in an object literal are optional if the name would be a legal JavaScript name and not a reserved word. So, quotes are required around `"firstname"`, but are optional around `first_name`. Commas are used to separate the pairs.

A property's value can be obtained from any expression, including another object literal. Objects can nest:

```
var flight = {
  airline: "Oceanic",
  number: 815,
  departure: {
  IATA: "SYD",
  time: "2004-09-22 14:55",
  city: "Sydney"
  },
  arrival: {
  IATA: "LAX",
  time: "2004-09-23 10:42",
  city: "Los Angeles"
  }
};
```

Retrieval

Values can be retrieved from an object by wrapping a string expression in a `[ ]` suffix. If the string expression is a string literal, and if it is a legal JavaScript name and not a reserved word, then the . notation can be used instead. The . notation is preferred because it is more compact and it reads better:

```
stooge[«first-name»]    // "Jerome"
flight.departure.IATA   // "SYD"
```

The undefined value is produced if an attempt is made to retrieve a nonexistent member:

```
stooge["middle-name"]  // undefined
flight.status          // undefined
stooge["FIRST-NAME"]   // undefined
```

The `||` operator can be used to fill in default values:

```
var middle = stooge["middle-name"] || "(none)";
var status = flight.status || "unknown";
```

Attempting to retrieve values from undefined will throw a `TypeError` exception. This can be guarded against with the `&&` operator:

```
flight.equipment                          // undefined
flight.equipment.model                    // throw "TypeError"
flight.equipment && flight.equipment.model // undefined
```

Update

A value in an object can be updated by assignment. If the property name already exists in the object, the property value is replaced:

```
stooge['first-name'] = 'Jerome';
```

If the object does not already have that property name, the object is augmented:

```
stooge['middle-name'] = 'Lester';
stooge.nickname = 'Curly';
flight.equipment = {
  model: 'Boeing 777'
};
flight.status = 'overdue';
```

Reference

Objects are passed around by reference. They are never copied:

```
var x = stooge;
x.nickname = 'Curly';
var nick = stooge.nickname;
// nick is 'Curly' because x and stooge are references to the same
   object
var a = {}, b = {}, c = {};
// a, b, and c each refer to a different empty object
a = b = c = {};
// a, b, and c all refer to the same empty object
```

Prototype

Every object is linked to a prototype object from which it can inherit properties. All objects created from object literals are linked to `Object.prototype`, an object that comes standard with JavaScript.

When you make a new object, you can select the object that should be its prototype. The mechanism that JavaScript provides to do this is messy and complex, but it can be significantly simplified. We will add a create method to the `Object` function. The create method creates a new object that uses an old object as its prototype. There will be much more about functions in the next chapter.

```
if (typeof Object.create !== 'function') {
Object.create = function (o) {
var F = function () {};
F.prototype = o;
return new F();
};
}
var another_stooge = Object.create(stooge);
```

The prototype link has no effect on updating. When we make changes to an object, the object's prototype is not touched:

```
another_stooge['first-name'] = 'Harry';
another_stooge['middle-name'] = 'Moses';
another_stooge.nickname = 'Moe';
```

The prototype link is used only in retrieval. If we try to retrieve a property value from an object, and if the object lacks the property name, then JavaScript attempts to retrieve the property value from the prototype object. And if that object is lacking the property, then it goes to its prototype, and so on until the process finally bottoms out with `Object.prototype`. If the desired property exists nowhere in the prototype chain, then the result is the undefined value. This is called delegation.

The prototype relationship is a dynamic relationship. If we add a new property to a prototype, that property will immediately be visible in all of the objects that are based on that prototype:

```
stooge.profession = 'actor';
another_stooge.profession // 'actor'
```

Reflection

It is easy to inspect an object to determine what properties it has by attempting to retrieve the properties and examining the values obtained. The `typeof` operator can be very helpful in determining the type of a property:

```
typeof flight.number    // 'number'
typeof flight.status    // 'string'
typeof flight.arrival   // 'object'
typeof flight.manifest  // 'undefined'
```

Some care must be taken because any property on the prototype chain can produce a value:

```
typeof flight.toString    // 'function'
typeof flight.constructor // 'function'
```

There are two approaches to dealing with these undesired properties. The first is to have your program look for and reject function values. Generally, when you are reflecting, you are interested in data, and so you should be aware that some values could be functions.

The other approach is to use the `hasOwnProperty` method, which returns true if the object has a particular property. The `hasOwnProperty` method does not look at the prototype chain:

```
flight.hasOwnProperty('number')       // true
flight.hasOwnProperty('constructor')  // false
```

Enumeration

The `for in` statement can loop over all of the property names in an object. The enumeration will include all of the properties – including functions and prototype properties that you might not be interested in – so it is necessary to filter out the values you don't want. The most common filters are the `hasOwnProperty` method and using `typeof` to exclude functions:

```
var name;
for (name in another_stooge) {
  if (typeof another_stooge[name] !== 'function') {
    document.writeln(name + ': ' + another_stooge[name]);
  }
}
```

There is no guarantee on the order of the names, so be prepared for the names to appear in any order. If you want to assure that the properties appear in a particular order, it is best to avoid the `for in` statement entirely and instead make an array containing the names of the properties in the correct order:

```
var i;
var properties = [
  'first-name',
  'middle-name',
  'last-name',
  'profession'
];
for (i = 0; i < properties.length; i += 1) {
  document.writeln(properties[i] + ': ' +
  another_stooge[properties[i]]);
}
```

By using `for` instead of `for in`, we were able to get the properties we wanted without worrying about what might be dredged up from the prototype chain, and we got them in the correct order.

Delete

The `delete` operator can be used to remove a property from an object. It will remove a property from the object if it has one. It will not touch any of the objects in the prototype linkage.

Removing a property from an object may allow a property from the prototype linkage to shine through:

```
another_stooge.nickname // 'Moe'
// Remove nickname from another_stooge, revealing
// the nickname of the prototype.
delete another_stooge.nickname;
another_stooge.nickname // 'Curly'
```

[D. Crockford, *JavaScript: The Good Parts*, O'Reilly, 2008]

QUESTIONS

a **Which JavaScript built-in elements are objects?**
b **How can a value from an object be retrieved?**
c **Are JavaScript objects passed by copy or by reference?**
d **What is the prototype chain?**

B2

Sviluppo di applicazioni web in linguaggio JavaScript

Oltre agli oggetti nativi e a quelli definiti dall'utente esistono gli oggetti forniti dallo *host*, che nel caso più comune è il browser che visualizza le pagine HTML. Gli oggetti forniti dallo *host* e le loro caratteristiche possono variare da un browser all'altro, anche se esiste un sottoinsieme comune di oggetti e caratteristiche di base.

1 Il DOM

Per poter modificare il contenuto di una pagina HTML è necessario che gli *script* possano accedere alla rappresentazione interna dei documenti. Oltre a questa caratteristica il browser offre agli *script* una collezione di oggetti che modellano vari aspetti di una pagina (DOM, *Document Object Model*) e del browser stesso.

Tutti gli oggetti esposti nel DOM possiedono alcune proprietà in comune. Le principali proprietà condivise sono riportate nella TABELLA 1.

TABELLA 1

Proprietà	Descrizione
id	Identificativo (dovrebbe essere univoco).
attributes	Vettore di attributi associati all'elemento.
childNodes	Vettore degli elementi (nodi) figli dell'elemento preso in considerazione (i figli di un elemento sono gli elementi contenuti nello stesso).
className	Identifica la classe di appartenenza dell'elemento.
innerHTML	Codice HTML che comprende il *tag* stesso e il suo contenuto.
nodeName	Nome del nodo corrente.
nodeType	Tipo del nodo corrente.
nodeValue	Valore del nodo corrente.
parentNode	Nodo padre del nodo corrente.
style	Stile applicato all'elemento corrente.
tabIndex	Ordine dell'elemento per la navigazione mediante tasto di tabulazione.

ESEMPIO Il codice seguente implementa una funzione, denominata *info*, che visualizza alcune informazioni tra quelle comuni a tutti gli oggetti esposti dal DOM; tale funzione viene invocata per due elementi della pagina HTML: il *body* e un *button*. La funzione è ricorsiva e invoca se stessa passando come argomento la proprietà *parentNode* e usando una lista non numerata per rendere evidente l'indentazione dei nodi che contengono l'elemento corrente.

▶

```
<!doctype html>
<html>
  <head>
    <script>
      function info(obj) {
        if (obj) {
          var s="";
          s = s + "className: "+obj.className+"<br>";
          s = s + "nodeName: "+obj.nodeName+"<br>";
          s = s + "nodeType: "+obj.nodeType+"<br>";
          s = s + "nodeValue: "+obj.nodeValue+"<br>";
          s = s + "tabIndex: "+obj.tabIndex+"<br>";
          s = s + "parentNode: {<ul><li>" +
            info(obj.parentNode) + "</li></ul>}<br>";
          s = s + "toString(): "+obj.toString() + "<br>";
          return s;
        }
      }
    </script>
  </head>
<body>
  <input type="button" id="b" name="b">
  <script>
    window.document.write(
      info(document.getElementsByTagName("body")[0])
    );
    window.document.write(
      info(document.getElementById("b"))
    );
  </script>
</body>
</html>
```

Esistono anche metodi comuni a tutti gli oggetti del DOM, i più utilizzati sono riportati nella TABELLA 2.

TABELLA 2

Metodo	Descrizione
addEventListener	Aggiunge un gestore per un determinato evento all'elemento a cui si applica.
removeEventListener	Elimina un gestore di evento esistente.
appendChild	Aggiunge un elemento in coda alla lista degli elementi contenuti nel nodo corrente.
cloneNode	Restituisce una copia dell'elemento a cui si applica.
dispatchEvent	Invia un evento.
getAttribute	Restituisce il valore dell'attributo specificato come argomento.
setAttribute	Ha due argomenti: attributo e nuovo valore. Imposta il nuovo valore all'attributo specificato come primo argomento.

ESEMPIO Il codice che segue modifica una pagina HTML contenente una tabella, colorando lo sfondo delle righe in posizione dispari facendo uso del metodo *getElementsByTagName* per recuperare tutte le righe e del metodo *setAttribute* per impostarne il colore di sfondo. ▶

```
<!doctype html>
<html>
  <body>
    <table style="border-width:1px; border-style:solid">
      <tr><td>Esempio</td></tr>
      <tr><td>Esempio</td></tr>
      <tr><td>Esempio</td></tr>
      <tr><td>Esempio</td></tr>
    </table>
    <script>
      var t = document.body.getElementsByTagName('tr');
      var colore = false;
      for (var i=0; i<t.length; i++) {
        if (colore)
        t[i].setAttribute("style", "background-color:yellow");
      colore = !colore;
    }
    </script>
  </body>
</html>
```

Nel seguito sono analizzati oggetti specifici esposti dal DOM, in particolare tutti i principali oggetti che il browser mette a disposizione per controllare o aggiornare l'ambiente di esecuzione.

ESEMPIO Per analizzare le proprietà e i metodi enumerabili di un oggetto, è possibile costruire una funzione che ne visualizza le principali caratteristiche:

```
<!doctype html>
<html>
  <script>
    function mostra(objName) {
      for (prop in objName) {
        let value=objName[prop];
        document.write("<b>" + prop + "</b>:" + value + "<br>");
      }
    }
  </script>
  <body onload="mostra(document)">
  </body>
</html>
```

La funzione *mostra* dell'esempio precedente itera le proprietà enumerabili dell'oggetto fornito come argomento e aggiunge, in fondo al documento corrente, una riga per ciascuna proprietà: ogni riga è composta dal nome della proprietà e dal suo valore.

OSSERVAZIONE Un browser potrebbe avere alcune proprietà enumerabili e un altro averle non enumerabili. Inoltre un browser potrebbe avere una proprietà (enumerabile o meno) e un altro non averla affatto. Pertanto si tratta di una tecnica utile, ma non esaustiva. Questa funzione è invece estremamente utile per indagare le diversità di implementazione tra browser diversi, e verrà utilizzata nel seguito per analizzare alcuni tra gli oggetti esposti.

1.1 L'oggetto *window*

Esiste un oggetto che è il genitore di tutti quelli messi a disposizione da un browser: l'oggetto ***window***, che modella la finestra corrente.

Di tale finestra è possibile conoscere le principali caratteristiche, come la sua dimensione, la sua posizione all'interno dello schermo e così via.

Nella tabella viene riportata una sintesi delle principali proprietà dell'oggetto *window*.

Proprietà o metodo	Descrizione
status	Permette di impostare messaggi sulla barra di stato del browser.
statusbar	Se la proprietà *visible* è impostata a `false` nasconde la barra di stato del browser, normalmente vale `true`.
locationbar	Se la proprietà *visible* è impostata a `false` nasconde la barra degli indirizzi del browser, normalmente vale `true`.
close( )	Chiude la finestra.
scrollbars	Se la proprietà *visible* è impostata a `false` nasconde le eventuali barre di scorrimento (orizzontale e verticale).
print( )	Avvia il processo di stampa della pagina che l'utente deve confermare.
toolbar	Valore booleano che indica se la barra degli strumenti del browser è visibile (`true`) o meno (`false`); proprietà in sola lettura per le finestre aperte.
screenLeft, screenTop	Rispettivamente, distanza dal bordo sinistro e da quello in alto, espressa in pixel, della finestra del browser rispetto allo schermo (alcuni browser).
screenX, screenY	Rispettivamente, distanza dal bordo sinistro e da quello in alto, espressa in pixel, della finestra del browser rispetto allo schermo (altri browser).

È possibile che una proprietà venga gestita in un modo da un browser e in modo diverso da un altro browser. Per realizzare pagine interoperabili è opportuno verificare la presenza di una proprietà inserendola in un test: se la proprietà non esiste, viene restituito `undefined` valutato come `false`; se la proprietà esiste, viene restituito un oggetto, o valore, valido, valutato come `true`.

ESEMPIO Il seguente *script* mostra le coordinate *X* e *Y* della finestra del browser rispetto al desktop:

```
<!doctype html>
<html>
  <script>
    function getX() {
      if (window.screenX)
        return window.screenX;
      else
        return window.screenLeft;
    }

    function getY() {
      if (window.screenY)
```

Finestre *pop-up*

Esistono metodi dell'oggetto *window* per aprire finestre *pop-up* (metodi: *open*, *close* e *moveTo*). Purtroppo alcune pratiche scorrette, come quella di aprire finestre *pop-up* a insaputa dell'utente per presentare messaggi pubblicitari, hanno portato molti browser all'integrazione di strumenti per il loro blocco. Chi si trova a creare applicazioni web deve tener presente tale possibilità e, pertanto, è consigliabile evitare una logica applicativa che faccia uso di *pop-up* separate dalla finestra principale. Inoltre l'uso di finestre *pop-up* può portare a logiche difficilmente replicabili su dispositivi dotati di browser semplificati (interfaccia di tipo SDI, *Single Document Interface*).