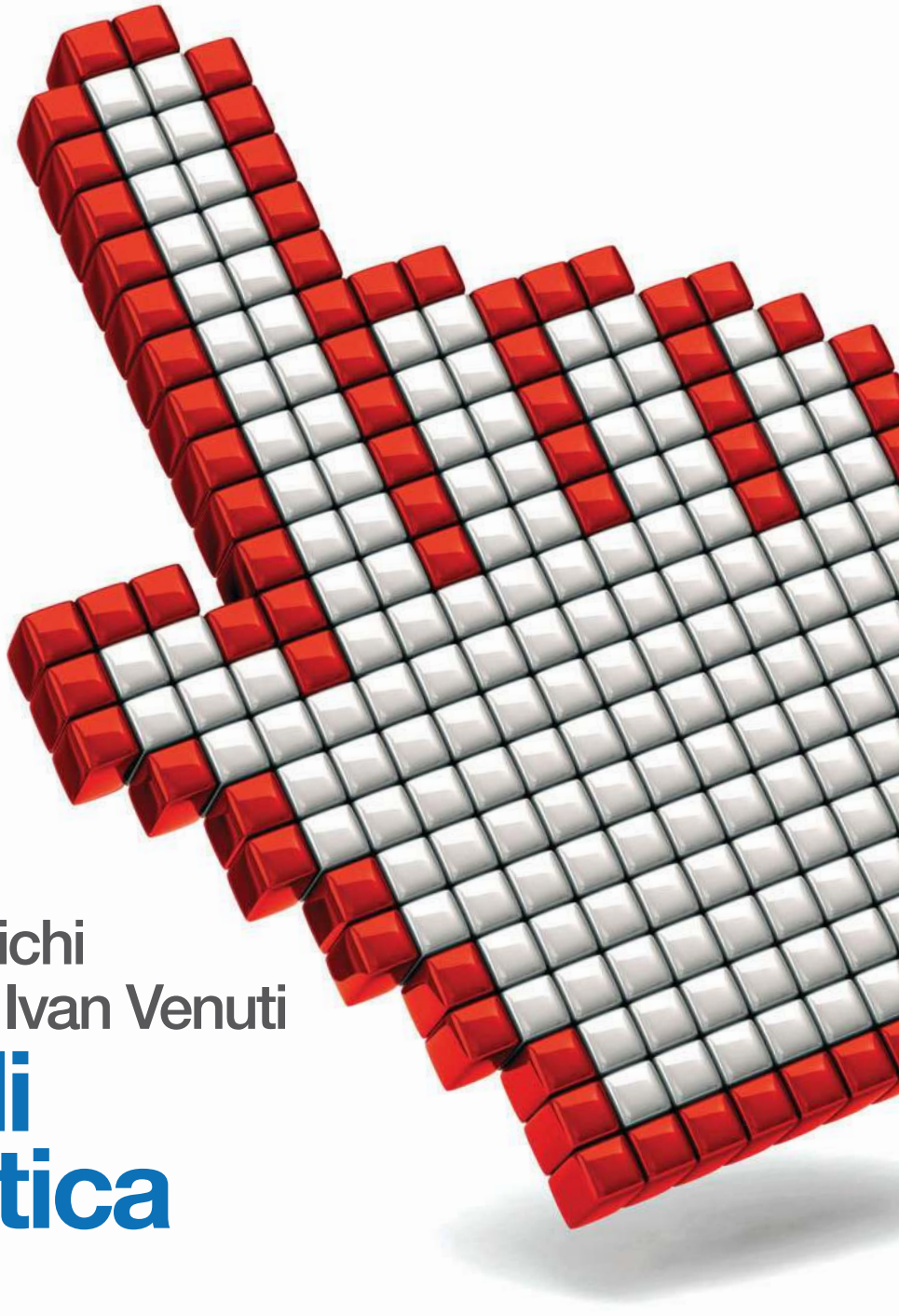
1 2 3 Idee per il tuo futuro



Fiorenzo Formichi
Giorgio Meini Ivan Venuti

Corso di informatica

per Informatica

Programmazione orientata agli oggetti e linguaggio Java
Pagine web con JavaScript

TECNOLOGIA ZANICHELLI

Fiorenzo Formichi
Giorgio Meini Ivan Venuti

Corso di informatica

per Informatica

Programmazione orientata agli oggetti e linguaggio Java
Pagine web con JavaScript

Realizzazione editoriale:
– Coordinamento redazionale: Matteo Fornesi
– Segreteria di redazione: Deborah Lorenzini
– Progetto grafico: Editta Gelsomini
– Collaborazione redazionale, impaginazione, disegni e indice analitico: Stilgraf, Bologna

Contributi:
– I capitoli della sezione B sono a cura di Ivan Venuti
– Rilettura dei testi in inglese: Roger Loughney

Copertina:
– Progetto grafico: Miguel Sal & C., Bologna
– Realizzazione: Roberto Marchetti
– Immagine di copertina: valdis torms/Shutterstock; Artwork Miguel Sal & C., Bologna

Prima edizione: gennaio 2012

L'impegno a mantenere invariato il contenuto di questo volume per un quinquennio (art. 5 legge n. 169/2008) è comunicato nel catalogo Zanichelli, disponibile anche online sul sito www.zanichelli.it, ai sensi del DM 41 dell'8 aprile 2009, All. 1/B.

File per diversamente abili
L'editore mette a disposizione degli studenti non vedenti, ipovedenti, disabili motori o con disturbi specifici di apprendimento i file pdf in cui sono memorizzate le pagine di questo libro. Il formato del file permette l'ingrandimento dei caratteri del testo e la lettura mediante software screen reader. Le informazioni su come ottenere i file sono sul sito www.zanichelli.it/diversamenteabili

Suggerimenti e segnalazione degli errori
Realizzare un libro è un'operazione complessa, che richiede numerosi controlli: sul testo, sulle immagini e sulle relazioni che si stabiliscono tra essi. L'esperienza suggerisce che è praticamente impossibile pubblicare un libro privo di errori. Saremo quindi grati ai lettori che vorranno segnalarceli.
Per segnalazioni o suggerimenti relativi a questo libro scrivere al seguente indirizzo:

lineazeta@zanichelli.it

Le correzioni di eventuali errori presenti nel testo sono pubblicati nel sito www.zanichelli.it/aggiornamenti

Zanichelli editore S.p.A. opera con sistema qualità
certificato CertiCarGraf n. 477
secondo la norma UNI EN ISO 9001:2008

Fiorenzo Formichi
Giorgio Meini Ivan Venuti

Corso di informatica

per Informatica

Programmazione orientata agli oggetti e linguaggio Java
Pagine web con JavaScript

TECNOLOGIA ZANICHELLI

Indice

SEZIONE A

Programmazione orientata agli oggetti e linguaggio Java

A1 Introduzione alla programmazione e alla progettazione orientate agli oggetti

A2 Il linguaggio di programmazione Java

A3 La programmazione orientata agli oggetti in Java

A4 Strutture dati

A5 Ereditarietà e polimorfismo

A6 Tipi generici e collezioni nel linguaggio Java

A7 Introduzione alle *Graphic User Interface* in Java

A8 Ambiente di sviluppo *NetBeans* e applicazioni Java con GUI *Swing*

A9 Gestione della concorrenza nel linguaggio Java

SEZIONE B

Pagine web con JavaScript

B1 Il linguaggio JavaScript

B2 JavaScript e il DOM, jQuery e Google *Maps*

B3 Strumenti per lo sviluppo di pagine web con JavaScript

I capitoli affiancati da questa icona sono disponibili, con chiave di attivazione, all'indirizzo: www.online.zanichelli.it/formichimeinicorsoinformatica

SEZIONE

A Programmazione orientata agli oggetti e linguaggio Java

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, …);
- 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:

- una astrazione sui dati mediante le sue caratteristiche (attributi);
- una 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*).

OSSERVAZIONE 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 impiegarli – 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 come 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, utilizzando strumenti come le pinze o i cacciavite, per ragioni sia di praticità sia di sicurezza: potremmo danneggiare il televisore o rimanere fulminati da una scarica elettrica!

La precedente osservazione 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 (i **dati**, o **attributi**), che 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 le operazioni definite 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. Il corretto 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 permette 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*: uno dei suoi principali contributi ha consistito nella partecipazione alla definizione dello *Unified Modeling Language* (UML) insieme a James Rambaugh e Ivar Jacobson (universalmente conosciuti come *los tres amigos*).

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.

ESEMPIO

Il concetto dell'*information hiding* è rappresentato graficamente nella 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.

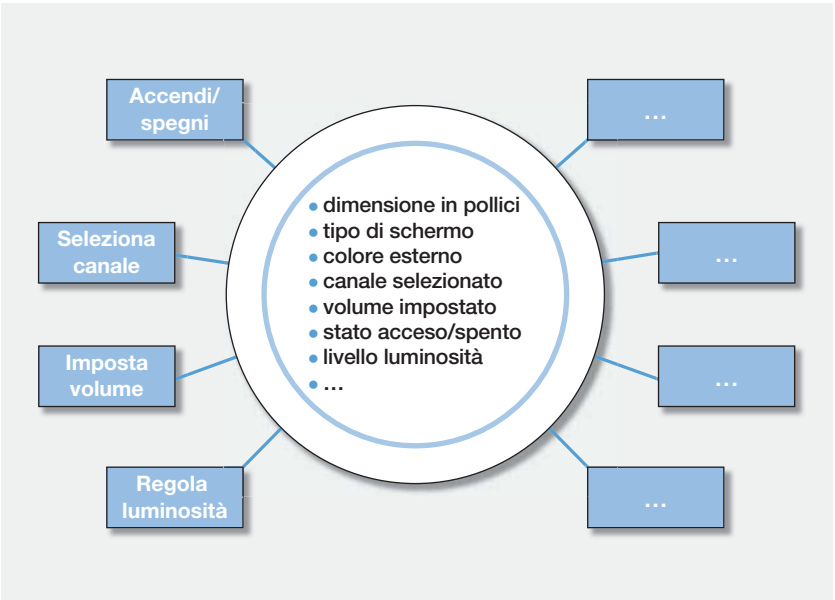


FIGURA **1**

OSSERVAZIONE 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 *alzaVolume* e *abbassaVolume* – 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.

ES. 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 ha come conseguenza 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

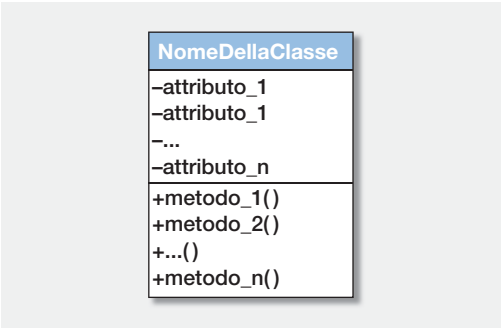
Unified Modeling Language

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, Jim Rumbaugh e Ivar Jacobson 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.

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.

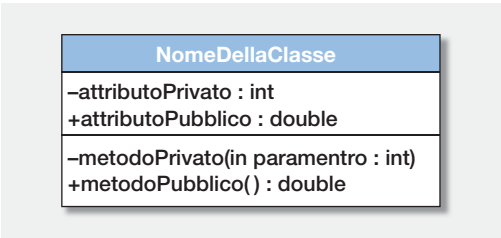
riscosso il maggior successo. Tra i vari formalismi grafici che UML mette a disposizione per descrivere i diversi aspetti di un sistema software a molteplici livelli di dettaglio, faremo riferimento al **diagramma delle classi** (*class diagram*) che, come è deducibile dal nome, fornisce una notazione 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:



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*.
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

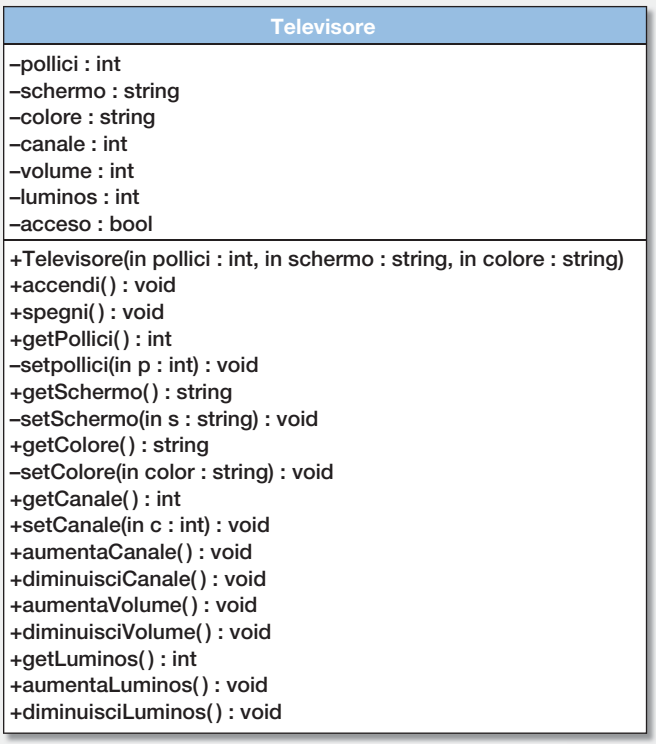


FIGURA **2**

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 stessa classe;
- 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.

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.

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

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.

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 **3** rappresenta il diagramma di sequenza che descrive l'acquisto di un oggetto in un negozio da parte di un cliente.

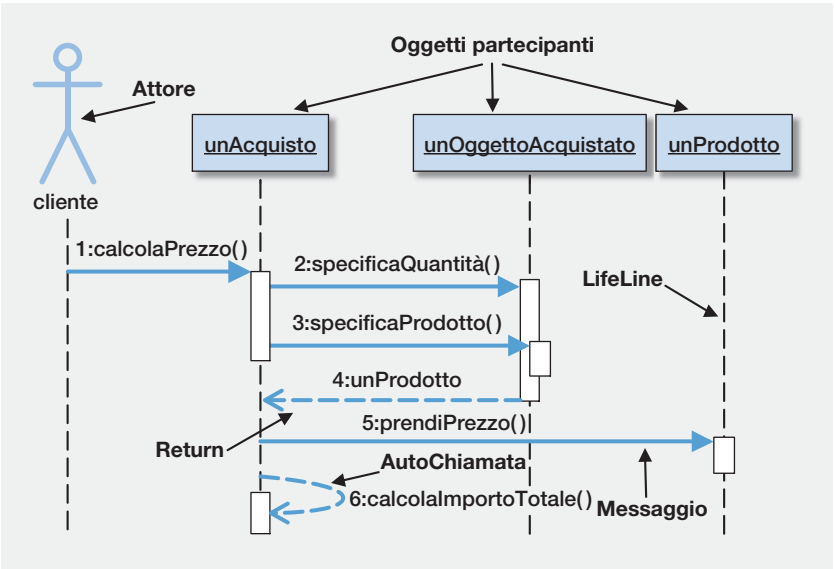


FIGURA **3**

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: ***oggetto : 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 (*life-line*)**: 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 colorato 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 compaiono 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 compaiono 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.

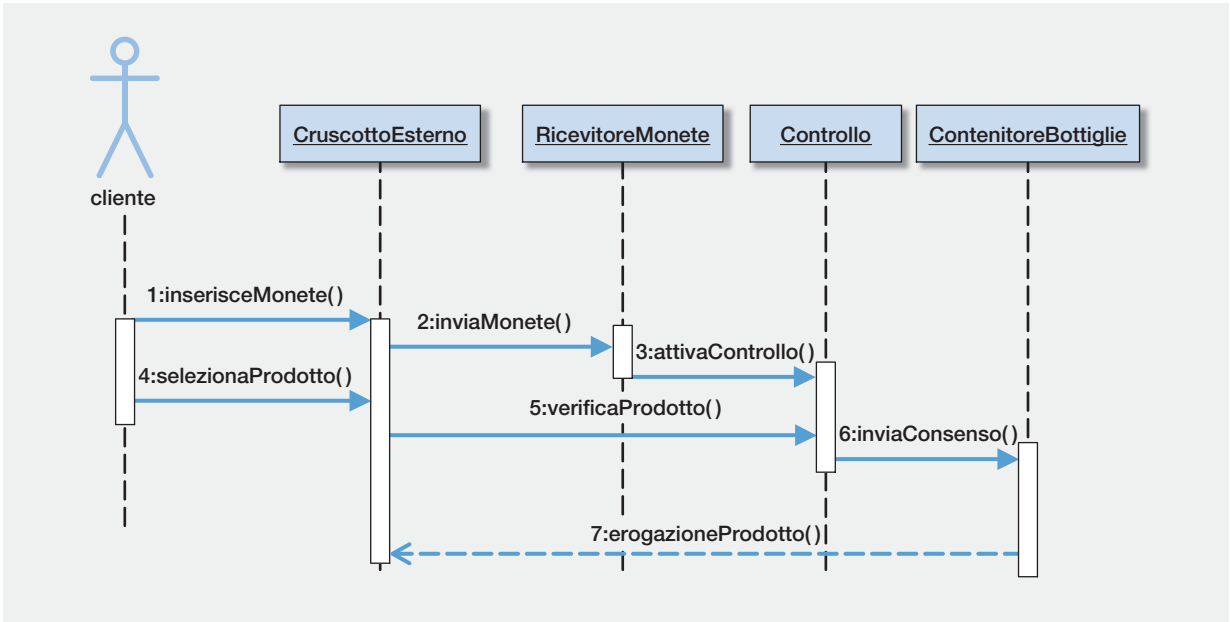


FIGURA **4**

Nel diagramma di sequenza possiamo individuare le seguenti azioni (FIGURA **4**):

- 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.

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 **5**):

- 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.

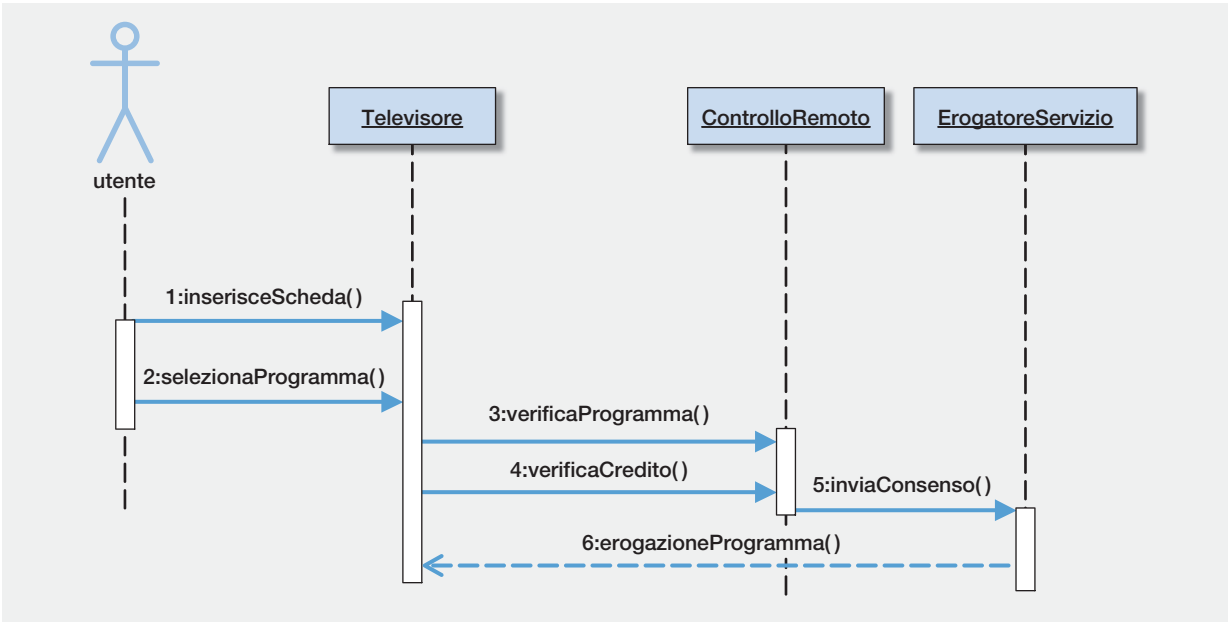


FIGURA **5**

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.

OSSERVAZIONE 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 **6** illustra la gerarchia delle classi ipotizzata nell'esempio precedente.

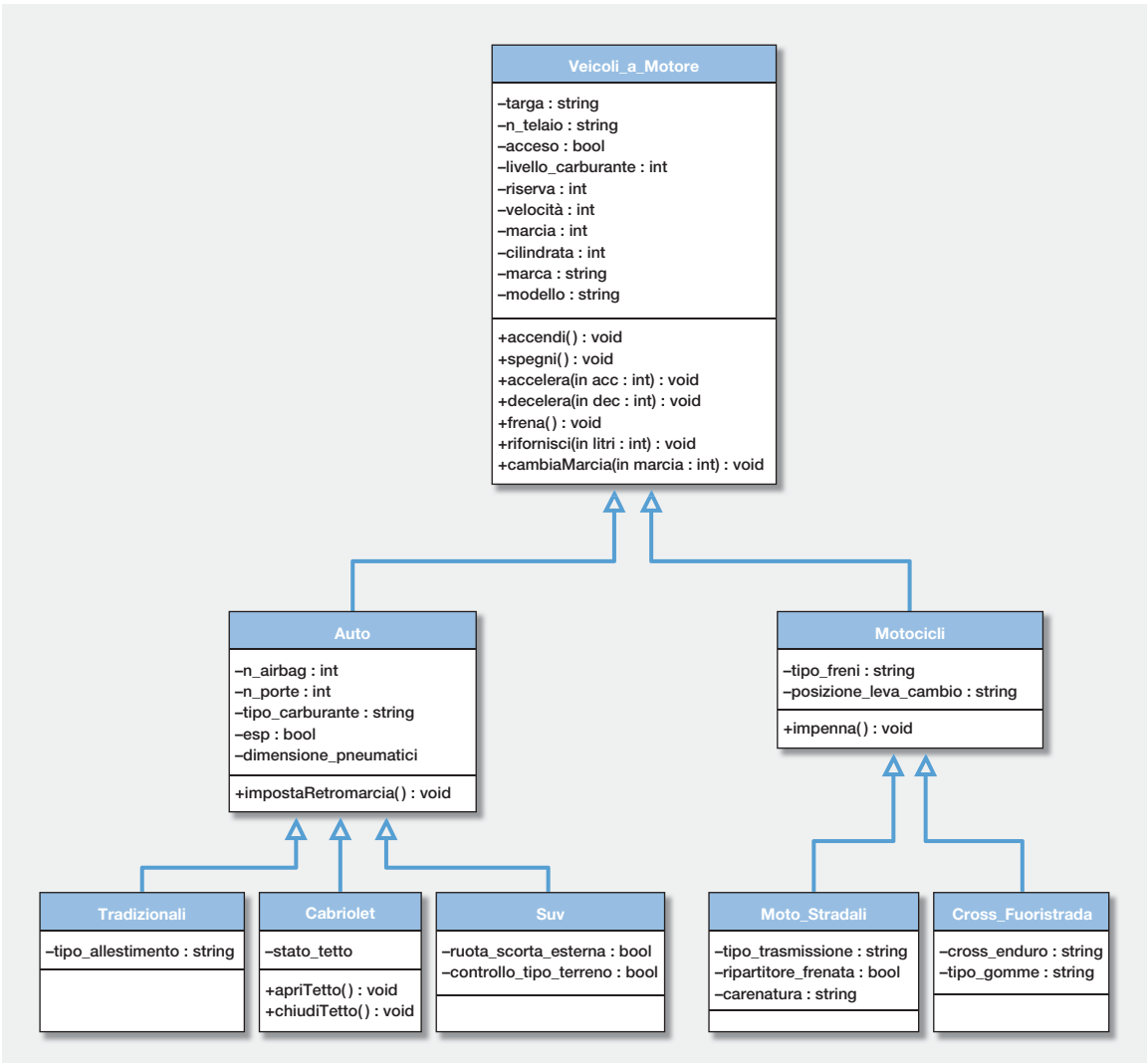


FIGURA **6**

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 superclassi: 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 da FIGURA **7**.
In questo caso i veicoli anfibi ereditano sia le caratteristiche dei veicoli terrestri sia quelle dei veicoli acquatici.

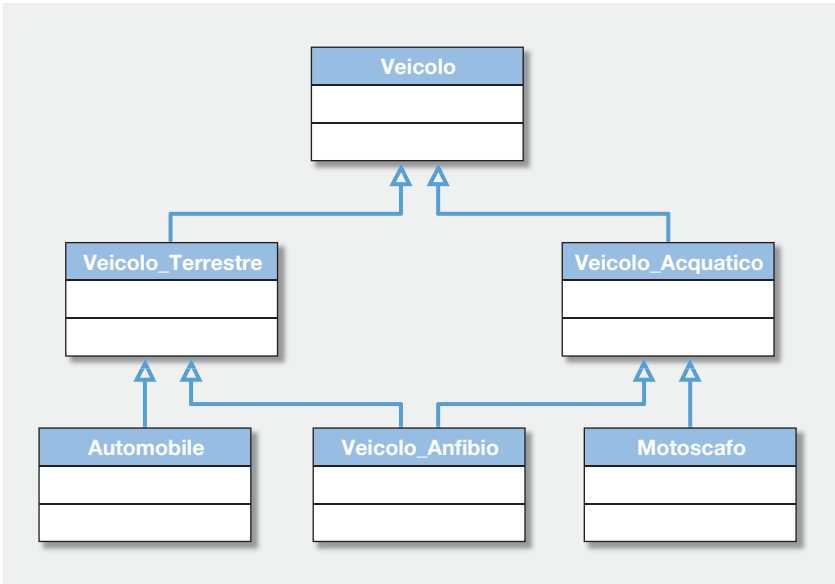


FIGURA **7**

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 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.

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 **8**.

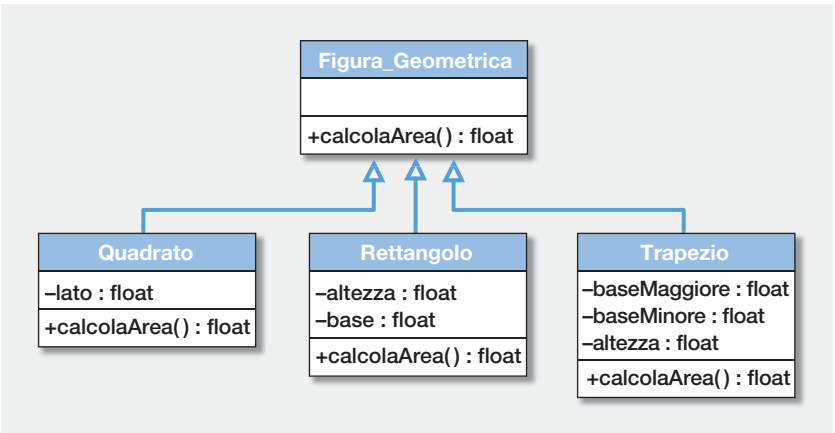


FIGURA **8**

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.

OSSERVAZIONE 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 definite **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 dichiarate.
Nei diagrammi UML una classe astratta viene indicata scrivendone la denominazione in carattere corsivo.

Interfacce

Alcuni linguaggi di programmazione orientati agli oggetti consentono di definire, oltre alle classi, anche le interfacce: una **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.

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, …).

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.

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 **9**).

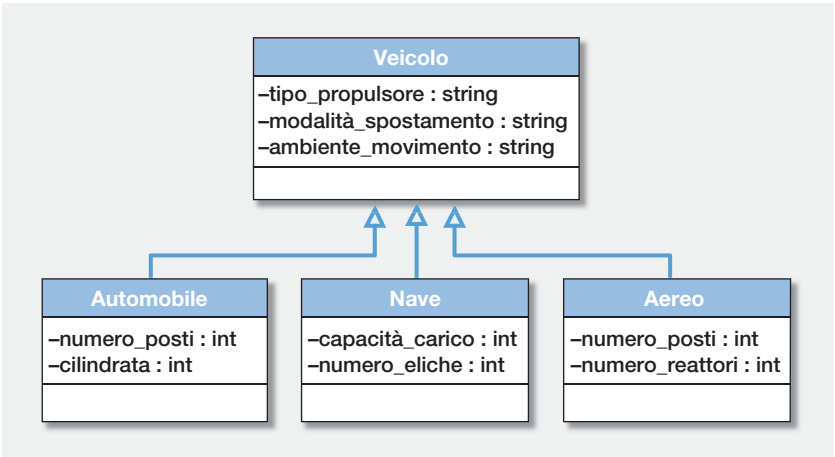


FIGURA **9**

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 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.
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.

ESEMPIO

Una 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 **10**).

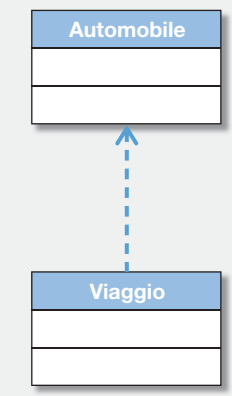


FIGURA **10**

OSSERVAZIONE L'associazione di dipendenza dell'esempio precedente indica che una 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 l'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

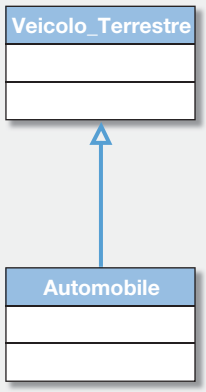


FIGURA **11**

L'ereditarietà tra classi realizzata dai linguaggi di programmazione OO è una 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.

OSSERVAZIONE 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 **11** rappresenta la classe *Veicolo_Terrestre* come generalizzazione della classe *Automobile* (la classe *Automobile* è di conseguenza una «specializzazione» della classe *Veicolo_Terrestre*).

5.3 Composizione

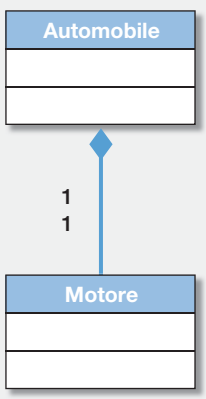


FIGURA **12**

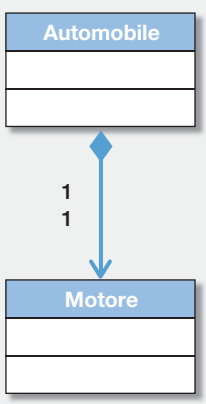


FIGURA **13**

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 **12** 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.

OSSERVAZIONE 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 la/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 **13**).

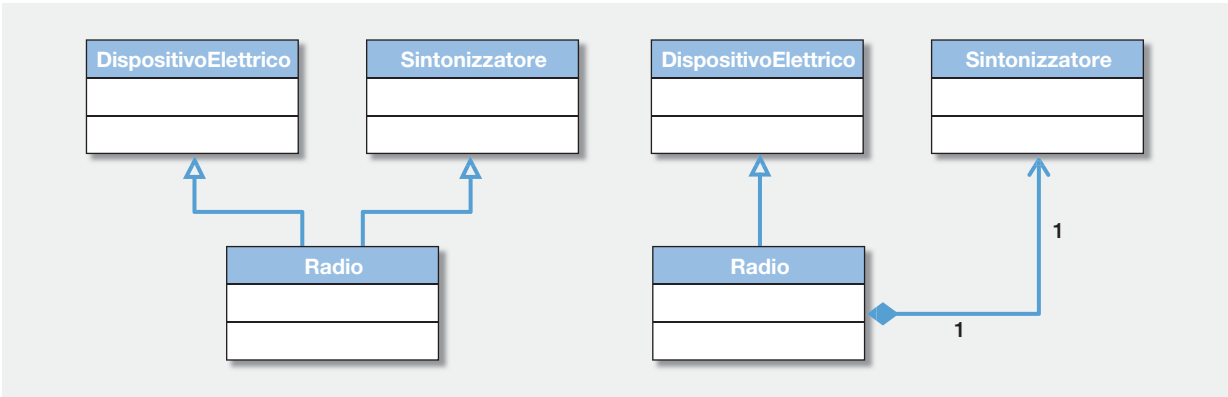


FIGURA **14**

OSSERVAZIONE 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 **14**).
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*.

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 **15** 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*.

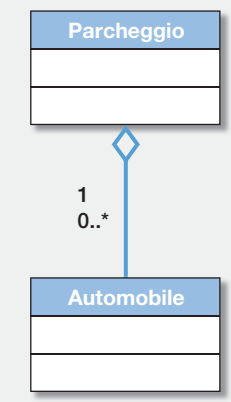


FIGURA **15**

OSSERVAZIONE 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.

Sintesi

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 (i *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 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 o mascherandone 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 classe deriva al più da una classe base, *multipla* se una classe deriva da due o più classi base.

Polimorfismo. È un concetto tipico della OOP e strettamente collegato alle dinamiche 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 in essa solo dichiarate. 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 classe derivata è al tempo stesso istanza della classe base (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.

QUESITI

1 La OOP è un paradigma di programmazione che permette la modellazione di sistemi reali basati …

A … essenzialmente su funzioni algoritmiche che implementano le funzionalità.

B … sulla definizione di oggetti software che interagiscono tra di loro attraverso lo scambio di messaggi.

C … sulla definizione di messaggi software che interagiscono tra di loro attraverso lo scambio di oggetti.

D Nessuna delle risposte precedenti.

2 Un ADT è …

A … un tipo di dato astratto definito completamente ma indipendente dalla sua implementazione software.

B … un tipo di dato astratto definito completamente dipendente dalla sua implementazione software.

C … un tipo di dato algoritmico non definito completamente e dipendente dalla sua implementazione software.

D Nessuna delle risposte precedenti.

3 Quali delle seguenti affermazioni circa un ADT sono vere?

A È un nuovo tipo di dato implementato dal softwarista che ne definisce sia il contenuto informativo sia le operazioni specifiche.

B Il contenuto informativo di un ADT è riferito alle operazioni specifiche a esso associate.

C Gli aspetti funzionali di un ADT sono relative alle operazioni specifiche a esso associate.

D Nessuna delle risposte precedenti.

4 L'interazione con un ADT avviene …

A … solo tramite un insieme di operazioni prestabilite detto interfaccia.

B … tramite un accesso diretto al suo contenuto informativo.

C … sia tramite l'interfaccia sia con un accesso diretto al suo contenuto informativo.

D Nessuna delle risposte precedenti.

5 L'*information hiding* è un meccanismo che nella OOP permette …

A … l'incapsulamento, separando di fatto il contenuto informativo di un ADT, che rimane nascosto, dalla sua interfaccia esposta verso l'esterno.

B … l'incapsulamento, separando di fatto il contenuto informativo di un ADT, esposto verso l'esterno, dalla sua interfaccia che rimane nascosta.

C … quando necessario, l'occultamento completo degli oggetti.

D Nessuna delle risposte precedenti.

6 Quali delle seguenti affermazioni circa l'incapsulamento sono vere?

A Rende invisibili all'esterno le scelte implementative di un ADT, eventualmente soggette a modifiche, proteggendo così il software che utilizza l'ADT dalle conseguenze dovute a tali cambiamenti.

B Rende visibili all'esterno in un'unica soluzione le scelte implementative di un ADT per facilitare il compito di chi sviluppa il software che utilizza l'ADT.

C Garantisce un corretto utilizzo del contenuto informativo di un ADT permettendone l'accesso solo tramite le funzioni di interfaccia, prevenendo così eventuali errori nella sua gestione da parte del codice che lo utilizza.

D Nessuna delle risposte precedenti.

7 Una classe è …

A … il meccanismo attraverso il quale vengono classificati vari tipi di ADT.

B … un modello formale per la descrizione di un certo tipo di oggetti definendone gli attributi, i metodi e le caratteristiche della sua interfaccia.

C … un modello formale per la descrizione di un certo tipo di oggetti definendone gli attributi, i metodi ma non le caratteristiche dell'interfaccia.

D Nessuna delle risposte precedenti.

8 **Quali delle seguenti affermazioni circa un oggetto nella OOP sono vere?**

A È l'istanza di una specifica classe.

B È un'entità autonoma con propri valori per gli attributi che lo caratterizzano.

C È un'entità autonoma con proprie operazioni che possono differire anche da quelle di altri oggetti simili.

D Nessuna delle risposte precedenti.

9 **L'UML è …**

A … un formalismo per l'implementazione del software che rende disponibili diversi tipi di istruzioni per descrivere i vari aspetti di un sistema OO.

B … un 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.

C … un formalismo per la progettazione e la documentazione del software che rende disponibili diversi tipi di linguaggi di programmazione per descrivere i vari aspetti di un sistema OO.

D Nessuna delle risposte precedenti.

10 **Quale tipo di diagramma UML è adatto a descrivere uno scenario di interazione tra oggetti?**

A *Class diagram*.

B *Use case diagram*.

C *Object diagram*.

D Nessuna delle risposte precedenti.

11 **L'ereditarietà è un meccanismo che permette …**

A … la definizione di nuove classi a partire da classi esistenti.

B … la creazione di nuovi oggetti a partire da oggetti esistenti.

C … la creazione di nuovi metodi a partire da metodi esistenti.

D Non prevede parametri.

12 **Quali delle seguenti affermazioni circa l'ereditarietà sono vere?**

A Permette di ereditare, nella classe derivata, solo attributi della classe base.

B Permette estendere la classe base con attributi e metodi della classe derivata.

C Permette di ereditare, nella classe derivata, attributi e metodi della classe base eventualmente estendendoli con nuovi attributi e metodi.

D Nessuna delle risposte precedenti.

13 **Il polimorfismo è un meccanismo che permette di …**

A … ridefinire nelle classi derivate i comportamenti della classe base, per ottenere metodi in grado di operare su oggetti di tipologia differente, lasciando scegliere all'ambiente di esecuzione il metodo corretto da utilizzare sulla classe base di cui un oggetto è istanza.

B … trattare gli oggetti istanza delle classi di una gerarchia derivata da una classe base come se fossero istanze della classe base stessa.

C … variare la tipologia di un oggetto a seconda dei contesti operativi in cui il software viene applicato.

D Nessuna delle risposte precedenti.

14 **Quali delle seguenti affermazioni circa una classe astratta sono vere?**

A È una classe in cui alcuni metodi sono solo dichiarati e non implementati.

B È una classe in cui non tutti gli attributi sono definiti.

C È una classe da cui non possono essere istanziati oggetti.

D È una classe che sicuramente verrà utilizzata come classe base da cui erediteranno altre classi.

15 **Un'associazione tra due classi …**

A … stabilisce le caratteristiche con cui gli oggetti di una classe si relazionano con quelli dell'altra.

B … permette di fondere le due classi in una nuova classe.

C … permette di creare oggetti che hanno caratteristiche di entrambe le classi.

D Nessuna delle risposte precedenti.

16 **Quali delle seguenti affermazioni sono vere rispetto all'associazione tra due classi?**

A Si parla di associazione statica quando gli attributi di una classe possono essere del tipo definito da un'altra classe.

B Si parla di associazione dinamica quando gli attributi di una classe possono essere del tipo definito da un'altra classe.

C Si parla di associazione dinamica quando sono gli scambi di messaggi tra oggetti che veicolano, con i loro parametri, le relazioni tra le classi.

D Si parla di associazione statica quando sono gli scambi di messaggi tra oggetti che veicolano, con i loro parametri, le relazioni tra le classi.

17 **Un'associazione d'uso è un'associazione in cui …**

A … le eventuali variazioni all'elemento fornitore possono avere influenza sull'elemento cliente.

B … le eventuali variazioni all'elemento cliente possono avere influenza sull'elemento fornitore.

C … le eventuali variazioni all'elemento fornitore o all'elemento cliente possono avere influenza reciproca sui due elementi.

D Nessuna delle risposte precedenti.

18 **La cardinalità di una associazione binaria tra classi esprime …**

A … il numero di oggetti che in totale partecipano all'associazione stessa.

B … il numero di oggetti che per ogni classe partecipano all'associazione stessa.

C … il numero di oggetti che per la sola classe dominante partecipano all'associazione stessa.

D Nessuna delle risposte precedenti.

19 **La generalizzazione è un'associazione …**

A … di tipo has-A che di fatto permette di realizzare l'ereditarietà tra classi.

B … di tipo is-A che di fatto permette di realizzare l'ereditarietà tra classi.

C … tra classi del tipo tutto-parti.

D Nessuna delle risposte precedenti.

20 **L'aggregazione è un'associazione …**

A … di tipo has-A che di fatto permette di realizzare l'ereditarietà tra classi.

B … di tipo is-A che di fatto permette di realizzare l'ereditarietà tra classi.

C … di tipo has-A che di fatto permette di realizzare un meccanismo tutto-parti in cui la classe tutto governa il periodo di vita delle classi «parti» e che a sua volta esiste solo in funzione delle classi «parti».

D Nessuna delle risposte precedenti.

21 **La composizione è un'associazione …**

A … di tipo has-A che di fatto permette di realizzare l'ereditarietà tra classi.

B … di tipo is-A che di fatto permette di realizzare l'ereditarietà tra classi.

C … di tipo has-A che di fatto permette di realizzare un meccanismo tutto-parti in cui la classe tutto governa il periodo di vita delle classi «parti» e che a sua volta esiste solo in funzione delle classi «parti».

D Nessuna delle risposte precedenti.

Chapter 18. UML

Unified Modeling Language (UML) is an object modelling specification language that uses graphical notation to create an abstract model of a system. The Object Management Group (http://www.uml.org/) governs UML. This modelling language can be applied to Java programs to help graphically depict such things as class relationships and sequence diagrams. Comprehensive information on UML is covered in UML Distilled, Third Edition, by Martin Fowler (Addison-Wesley).

18.1 Class Diagrams

A class diagram represents the static structure of a system, displaying information for classes and relationships between them. The individual class diagram is divided into three compartments: name, attributes (optional), and operations (optional); see Figure 18.1 and the example that follows it.

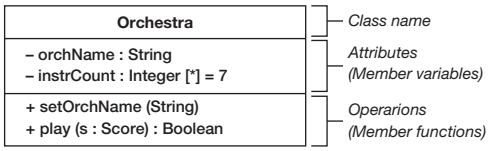


Figure 18.1. Class diagram.

```
// Corresponding code segment
  class Orchestra { // Class Name
    // Attributes
    private String orchName;
    private Integer instrCount = 7;
    // Operations
    public void setOrchName(String name) {...}
    public Boolean play(Score s) {...}
  }
```

[...]

18.7 Class Relationships

Class relationships are represented by the use of connectors and class diagrams; see Figure 18.6. Graphical icons, multiplicity indicators, and role names may also be used in depicting relationships.

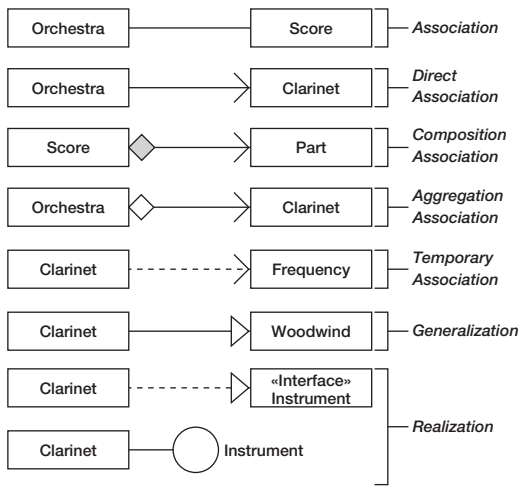


Figure 18.6. Class relationships.

role
ruolo

relationship
relazione

whole-part
intero-parte

woodwind
strumento musicale a fiato

graphically depict
rappresentazione grafica

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, 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    alzaVolume() {
    if (volume<50) volume++;
  }
  public  void    abbassaVolume() {
    if (volume>0) volume--;
  }
  public  void    aumentaLuminosita() {
    if (luminosita<80) luminosita ++;
  }
  public  void    diminuisceLuminosita() {
    if (luminosita>0) luminosita--;
  }
}
```

Anche se non abbiamo ancora introdotto le caratteristiche del Java, confrontando questo codice col 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 Microsystem (in seguito acquisita da Oracle Corporation) a partire dalle ricerche effettuate presso l'università di Stanford in California agli inizi degli anni '90 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.

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 *bytecode* (il formato eseguibile dei programmi Java) è in esadecimale CAFEBABE, cioè letteralmente «ragazza del caffè», forse un omaggio alla cameriera del caffè.

Bytecode

Il *bytecode* 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 *bytecode* 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 *bytecode*, 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 *bytecode* 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.

- **Portabilità**. Il motto dei progettisti di Java è stato «*Write once, run everywhere*» («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 *bytecode*, che viene interpretato dalla JVM. Ogni piattaforma hardware/software ha una propria JVM specifica, per cui il *bytecode* può essere indifferentemente eseguito su ciascuna piattaforma perché per ognuna di esse è la macchina virtuale a farsi carico della sua esecuzione.

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 *bytecode* con estensione «.class» che può essere eseguito su qualsiasi piattaforma hardware/software se dotata della specifica JVM.

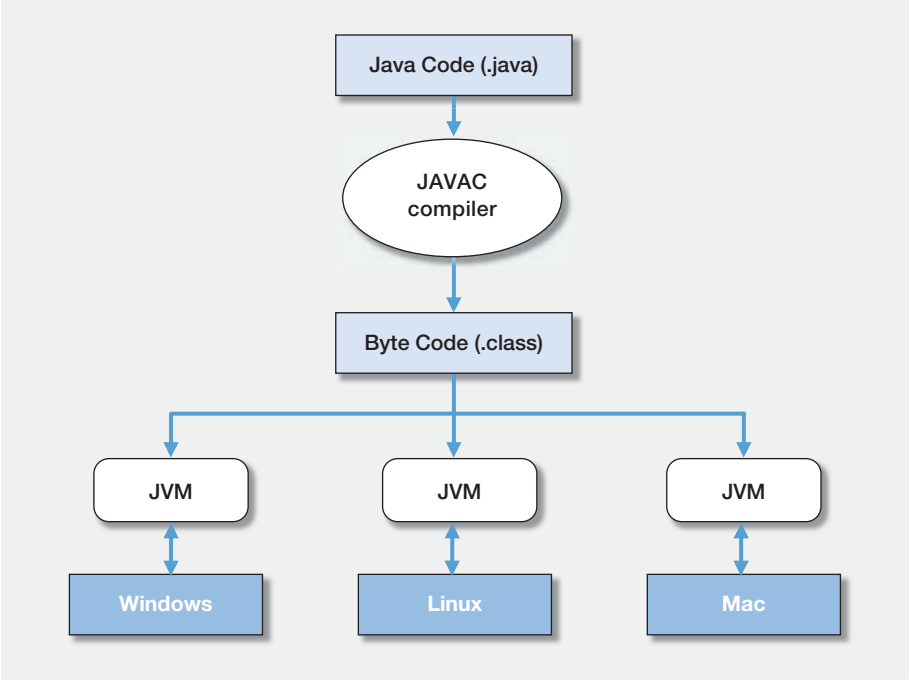


FIGURA **1**

In pratica la piena portabilità è un obiettivo tecnicamente difficile da raggiungere e il fatto che tecnologia del 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*» («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 che 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.

ESEMPIO

Le *applet* Java sono applicazioni software eseguite dal browser che ne scarica il *bytecode* da un server web remoto. La JVM esegue il *bytecode* scaricato in una modalità sicura (*sandbox*) con funzionalità ridotte, in modo da impedire un eventuale comportamento malevolo. La possibilità di eseguire alcuni tipi di operazioni (come la creazione, modifica e cancellazione di file) deve sempre essere approvata dall'utente. Le *applet* non hanno avuto una grande diffusione, a causa del fatto che per la loro esecuzione è necessario che il computer su cui esse devono essere eseguite disponga della JVM. Inoltre ormai si preferiscono applicazioni che prevedono il cosiddetto *thin-client* («client leggero») che interagisce con il codice remoto utilizzando solo il browser.

- **Multithreading nativo**. Il linguaggio di programmazione Java è stato uno dei primi a definire un modello di *multithreading* 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 *multicore*, 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 de-

gli 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 dal 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`).

OSSERVAZIONE 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 (*bytecode*) 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 *bytecode*.

OSSERVAZIONE Il file «Hello.class» è in formato binario, ma il *bytecode* che contiene può essere visualizzato mediante il comando:

```
>javap -c Hello
```

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

```
>java Hello
```

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

```
Hello, world!
```

OSSERVAZIONE 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.

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

- un insieme di istruzioni (*bytecode*);
- 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 del metodi.

Bytecode: l'insieme di istruzioni della macchina virtuale è ottimizzato in modo che risulti piccolo e compatto; è stato infatti progettato per scaricare il codice eseguibile su una rete di computer ed è stata sacrificata una maggiore velocità di interpretazione a favore di una riduzione dello spazio.

JRE e JDK

Per eseguire programmi Java su un computer è sufficiente installare il JRE (*Java Runtime Environment*) che comprende la JVM (*Java Virtual Machine*) 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.

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 *bytecode*.

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 *bytecode* dei metodi delle classi del programma Java in esecuzione.

OSSERVAZIONE 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.alzaVolume();
    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**.

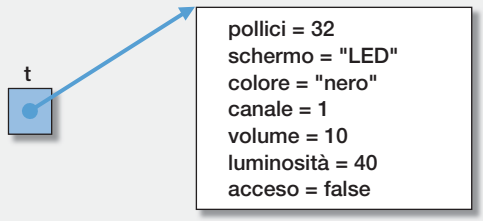


FIGURA **2**

OSSVAZIONE 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");
```

OSSERVAZIONE 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**.

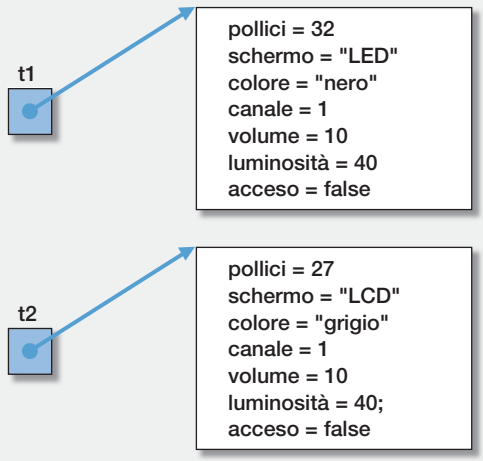


FIGURA **3**

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

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

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

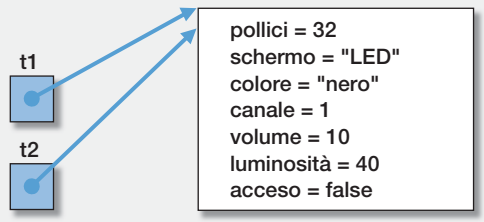


FIGURA **4**

OSSERVAZIONE 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 alcuno 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;
```

OSSERVAZIONE 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.alzaVolume();
…
```

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.alzaVolume();
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 *alzaVolume*.

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 *bytecode* 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*:
  - *package* delle librerie di base;
  - *package* delle estensioni;
  - *package* delle librerie aggiuntive;
  - *package* dell'applicazione;
- definizione della classe:
  - attributi pubblici;
  - attributi privati;
  - attributi protetti;
  - altri 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;
…
```

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

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

In Java è legale definire una classe priva di attributi e di metodi come la seguente:

```
public class Test {
  // attributi
  // costruttori
  // metodi
}
```

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 dalla parola chiave **final**.

ESEMPIO

Il valore di π può essere così definito:

```
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. 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 una eventuale omonimia di un parametro o variabile locale che lo nasconde;
- accessibilità stabilita tramite i modificatori **private**, **protected**, **public** e **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	SI	SÌ	NO
`public`	SÌ	SI	SÌ	SÌ
	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.

ESEMPIO

Il livello più restrittivo è `private`: un membro privato è accessibile solo dal codice dei metodi della stessa classe in cui è definito.

```
public class Alfa {
   private int attributoPrivato;
   private void metodoPrivato() {
      …
      attributoPrivato = 10; // legale
      …
   }
}
```

Solo gli oggetti di classe *Alfa* possono accedere all'attributo *attributoPrivato* e invocare il metodo *metodoPrivato*, cosa non permessa agli oggetti istanza della classe *Beta*:

```
public class Beta {
   void metodo() {
      Alfa a = new Alfa();
      a.attributoPrivato = 10; // illegale
      a.metodoPrivato(); // illegale
   }
}
```

Compilando il codice precedente si otterrebbe una segnalazione di errore.

ESEMPIO

Il modificatore di accesso **protected** permette al codice dei metodi della classe stessa, delle eventuali classi derivate e delle classi dello stesso *package* di accedere ai membri della classe: si deve utilizzare questa modalità di accesso quando non si vuole permettere l'accesso ai membri di una classe da parte di classi che non hanno alcuna relazione con essa. Se la seguente classe *Alfa* fa parte del *package greco*

```
package greco;

public class Alfa {
  protected int attributoProtetto;
  protected void metodoProtetto() {
    …
    …
    …
  }
}
```

i metodi della classe *Gamma* dello stesso *package* possono accedere ai suoi membri protetti:

```
package greco;

public class Gamma {
  void metodo() {
    Alfa a = new Alfa();
    a.attributoProtetto = 10; // legale
    a.metodoProtetto(); // legale
  }
}
```

I metodi di una classe *A* che deriva da *Alfa* e appartiene a un diverso *package* possono accedere ai membri protetti che *A* eredita da *Alfa*, ma non possono accedere ai membri protetti di un oggetto di classe *Alfa*:

```
package latino;
import greco.*;

public class A extends Alfa {
  void metodo() {
    Alfa a = new Alfa();
    a.attributoProtetto = 10; // illegale
    a.metodoProtetto(); // illegale
    A a = new A();
    a.attributoProtetto = 10; // legale
    a.metodoProtetto(); // legale
  }
}
```

ESEMPIO

Il livello meno restrittivo per l'accesso ai membri di una classe è **public**. I metodi di qualsiasi classe a qualsiasi *package* appartenga può accedere ai membri con questo livello di accesso di un'altra classe.

```
package greco;

public class Alfa {
  public int attributoPubblico;
```

▶

```
  public void metodoPubblico() {
    …
    …
    …
  }
}

package latino;
import greco.*;

public class Beta {
  void metodo() {
    Alfa a = new Alfa();
    a.attributoPubblico = 10; // legale
    a.metodoPubblico(); // legale
  }
}
```

ESEMPIO

Il livello *package* viene implicitamente assunto quando non si dichiara esplicitamente un modificatore d'accesso; esso permette solo alle classi dello stesso *package* di accedere ai membri di questo tipo:

```
package greco;

public class Alfa {
  int attributo;
  void metodo() {
    …
    …
    …
  }
}

package greco;

public class Beta {
  void metodo() {
    Alfa a = new Alfa();
    a.attributo = 10; // legale
    a.metodo(); // legale
  }
}

package latino;

public class B {
  void metodo() {
    Alfa a = new Alfa();
    a.attributo = 10; // illegale
    a.metodo(); // illegale
  }
}
```

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. Una 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.

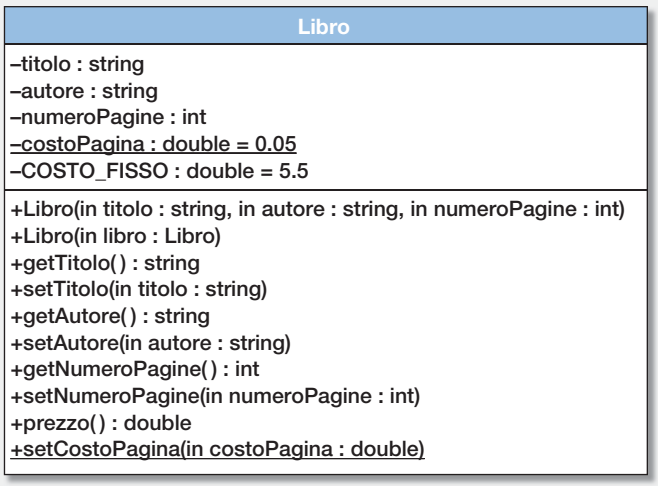


FIGURA **5**

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.titolo = 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 un classe, ma per invocarli è sufficiente premettere il nome della classe al loro nome.

I metodi statici hanno comunque una 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 – dovendo necessariamente essere invocato per la costruzione di oggetti – il suo livello di accessibilità deve essere `public`.
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 (*);

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

(*) 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;
}
```

4 La struttura di base di una classe e il metodo *main*

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

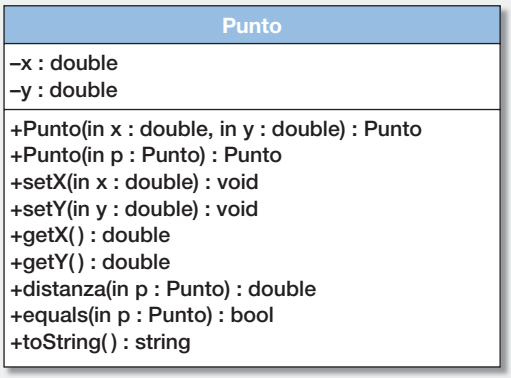


FIGURA **6**

Java Enterprise e POJO

La diffusione del linguggio 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 Java Bean*).
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.

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="+p2.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=(2.0;2.0)
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).

OSSERVAZIONE 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 buona 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**).

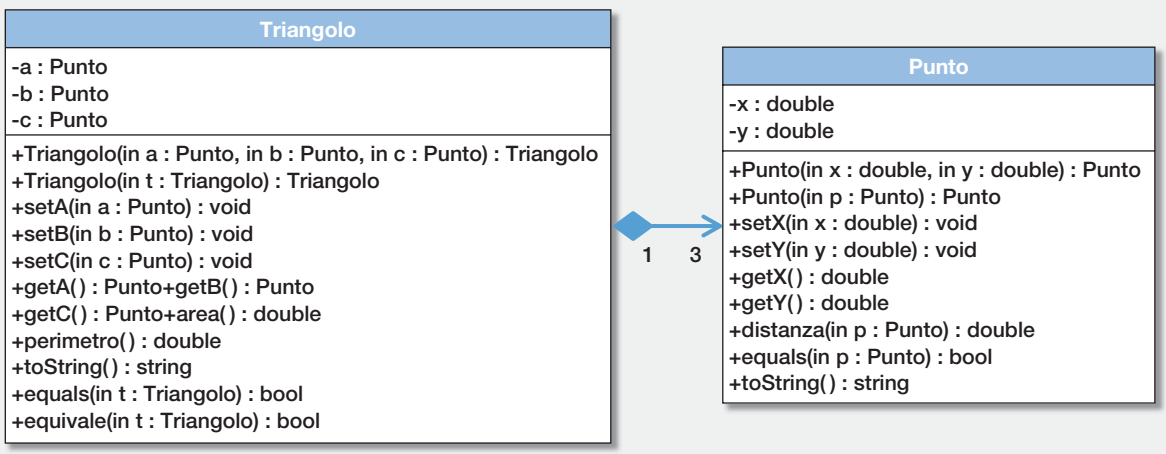


FIGURA **7**

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    this.a = a;
12    this.b = b;
13    this.c = c;
14  }
15
16  public Triangolo (Triangolo t) {
17    a = t.a;
18    b = t.b;
19    c = t.c;
20  }
21
22  public void setA(Punto a){this.a = a;}
23  public void setB(Punto b){this.b = b;}
24  public void setC(Punto c){this.c = 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 E' 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(t5))
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 E' 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
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 sono equivalenti
```

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

- Viene utilizzata la classe *Punto* prima definita per rappresentare per ogni oggetto di classe *Triangolo* i tre punti che rappresentano i suoi vertici (4-5).
- Esistono due diversi costruttori in *overloading* (10-14, 16-20): il primo istanzia 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.
- Sono definiti metodi *getter* e *setter* per i tre attributi dichiarati di tipo privato (22-24, 26-28).
- 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).
- 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 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.

- 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 convenzioni da seguire per lo stile del codice sorgente: in molti casi si tratta di consuetudini che, essendo molto diffuse tra gli sviluppatori, facilitano la lettura del codice, ma in alcuni casi specifici – pur trattandosi di regole che la sintassi del linguaggio non impone – esse sono assunte dagli ambienti di sviluppo e il loro mancato rispetto ne comporta l'erroneo funzionamento. Lo scopo di una convenzione per lo stile di scrittura del codice sorgente è quello di rendere maggiormente leggibile un programma: applicare uno stile coerente nello sviluppo di una applicazione software riduce lo sforzo necessario alla sua comprensione e il tempo impiegato – e, di conseguenza, il costo necessario – per le inevitabili modifiche.
In accordo con la struttura gerarchica degli elementi di un programma Java, vedremo lo stile da adottare per i *package*, le classi, gli attributi e i metodi. Come premessa enunciamo alcune regole generali comuni ricordando che il linguaggio Java – così come il linguaggio C/C++ – è *case sensitive* (cioè distingue i caratteri maiuscoli da quelli minuscoli), per cui è necessario fare attenzione a come sono scritte sia le parole chiave sia gli identificatori. I nomi degli identificatori dovrebbero essere sempre individuati in modo da:

- descrivere lo scopo di 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 mutuati da nomi che sono già utilizzati nel contesto d'uso: a questo proposito le librerie del linguaggio Java e i *pattern* di progettazione e di programmazione rappresentano buoni riferimenti;
- 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*).

OSSERVAZIONE Molti sviluppatori preferiscono – e molte aziende software impongono – utilizzare nomi inglesi per gli identificatori. Non è un vezzo: se le persone coinvolte in un progetto software sono di varia nazionalità, lo scopo dell'elenco di avvertenze riportato sopra viene completamente vanificato dalla mancata comprensione della lingua.

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.

5.1 *Package*

Il nome di un *package* dovrebbe essere univoco, scritto con soli caratteri minuscoli; per garantire l'univocità universale, se si dispone di un dominio Internet, il nome dovrebbe includerlo iniziando dal livello più alto (.com, .it, …), seguito dal nome del dipartimento e/o del progetto ed, infine, dal il nome assegnato al *package*:

```
<dominio_organizzazione>.<dipartimento>.
        <progetto>.<nome_package>
```

ES.

```
package com.sun.internal.java.language
package it.galileilivorno.abacus.informatica.esempi
```

La sezione del codice relativa all'importazione dei *package* dovrebbe prevederne il raggruppamento secondo la seguente sequenza:

- *package* della libreria di base (*java.**);
- *package* delle estensioni standard (*javax.**);
- *package* delle librerie aggiuntive (esempio: *org.w3c.dom.**);
- *package* dell'applicazione che si sta sviluppando.

5.2 Classi

Il nome di una classe dovrebbe sempre iniziare con una lettera maiuscola ed essere formato da un insieme di sostantivi, ognuno dei quali inizia con una lettera maiuscola e, dato che rappresentano «cose», andrebbero resi come nomi veri e propri.

ESEMPIO

```
class Televisore {};
class TriangoloIsoscele {};
class ProdottoVendutoOnline extends ProdottoVenduto {};
```

5.3 Attributi

I nomi degli attributi dovrebbero essere costruiti concatenando nomi e aggettivi, ognuno dei quali, escluso il primo, inizia con una lettera maiuscola. Per migliorare la leggibilità del codice è preferibile scrivere su una singola riga la dichiarazione di un solo attributo piuttosto che di una lista di attributi dello stesso tipo.

ESEMPIO

```
protected String descrizione;
private int identificatoreProdotto;
public String codiceBarreProdotto;
```

Gli attributi costanti sono qualificati dalla parola chiave `final` e il loro nome dovrebbe essere scritto tutto in maiuscolo, separando le parole con il simbolo «_».

ES.

```
final int MASSIMA_DIMENSIONE = 255;
final double PI_GRECO = 3.141592;
```

5.4 Metodi

I nomi dei metodi dovrebbero essere dei verbi che identificano l'operazione eseguita dal metodo, seguiti dal complemento; ogni parola successiva alla prima inizia con una lettera maiuscola.

ES.

```
public double calcolaCostoScontato() {…}
public void impostaAliquotaIva(double iva) {…}
```

OSSERVAZIONE 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 il prefisso *get* o *set* seguito 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() {…};
```

I parametri di un metodo dovrebbero avere nomi significativi: le stesse convenzioni che si applicano agli attributi si utilizzano per i nomi dei parametri dei metodi.

ESEMPIO

```
public void impostaDati(int codiceNumerico, String
  descrizione) {…}
public void aggiungiCliente(Cliente cliente) {…}
public void aggiungiMessaggio(Utente utente, Messaggio
  messaggio) {…}
```

Se i parametri di un metodo sono numerosi, è bene suddividerne l'elenco su più righe di codice indentate.

ESEMPIO

```
public void setProdotto(String categoria, String descrizione,
                        float costo, int quantita,
                        boolean disponibile) {…}
```

Per aumentare la leggibilità del codice, il valore di ritorno di un metodo dovrebbe essere contenuto in una variabile dichiarata localmente al metodo, il cui valore viene restituito al termine del metodo; per quanto possibile non dovrebbero esistere altre istruzioni di ritorno oltre quella posta alla fine del metodo.

ESEMPIO

```
public float getImporto() {
  float importo = 0.0;
  importo = prezzo-prezzo*sconto/100;
  importo += importo*tabellaIVA.getAliquotaIVA()/100;
  return importo;
}
```

Per quanto riguarda il codice dei metodi si dovrebbero infine 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 «;».
- Nei cicli si dovrebbe dichiarare la variabile contatore internamente per confinare l'area di attenzione richiesta allo sviluppatore.

ESEMPIO

```
for (int i = 0; i<n; i++) {
   …
   …
   …
}
```

- Le variabili locali dei metodi devono avere una visibilità più ristretta possibile; a questo scopo si dichiarano le variabili appena prima del loro utilizzo, per limitare l'area di attenzione richiesta allo sviluppatore.

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`. ▶

▶ TABELLA 2

`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`

Ogni classe *wrapper* ha un unico attributo del tipo primitivo che essa incapsula (la classe *Integer* ha un attributo di tipo `int`, la classe *Long* ha un attributo di tipo `long`, …). Esse sono state progettate e realizzate in modo da essere «immutabili»; gli oggetti istanza della classi *wrapper* assumono il valore dell'attributo incapsulato al momento della creazione tramite il costruttore e questo valore non può essere successivamente modificato, ma solo acquisito invocando uno specifico metodo. Inoltre le classi *wrapper* espongono molti metodi statici di utilità, per esempio per convertire stringhe di caratteri in valori numerici e viceversa.

ESEMPIO

L'esecuzione del seguente codice Java

```
public class EsempioInteger {
  public static void main(String args[]) {
    int n = 100;
    Integer numero = new Integer("10");
    n = n*numero.intvalue();
    System.out.println("N="+n);
  }
}
```

visualizza come risultato

```
N=1000
```

Uno dei costruttori della classe `Integer`, infatti, consente di valutare numericamente il contenuto di una stringa di caratteri (in questo caso «10» che fornisce il valore numerico 10), mentre il metodo `intvalue()` restituisce il valore di tipo **`int`** dell'attributo interno dell'oggetto `numero`.
Tra i metodi statici di utilità della classe `Integer` vi è `valueOf()`, che restituisce un oggetto di classe `Integer` inizializzato con un valore di tipo **`int`**:

```
Integer numero = Integer.valueOf(123);
```

e `parseInt()`, che restituisce un valore di tipo **`int`** a partire da una stringa di caratteri:

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

Quest'ultima tecnica è la più utilizzata per trasformare una stringa di caratteri in un valore numerico ed è sostanzialmente diversa da quella presente nel codice del metodo di esempio, in quanto non viene istanziato nessuno oggetto di tipo `Integer`, ma viene semplicemente utilizzato un metodo statico della classe `Integer`.
Infine il metodo statico `toString()` della classe `Integer` consente di trasformare in una stringa di caratteri un valore numerico di tipo **`int`**:

```
String string = Integer.toString(123);
```

OSSERVAZIONE Il codice dell'esempio precedente è relativo alla classe *Integer*, ma funzionalità analoghe sono offerte da tutte le classi *wrapper* dei tipi primitivi.

OSSERVAZIONE Nel codice dell'esempio precedente il valore numerico della variabile *n* viene implicitamente convertito in una stringa concatenata con il prefisso «N»; questo comportamento poteva essere esplicitato come nella seguente riga di codice:

```
System.out.println("N="+Integer.toString(n));
```

Ma l'operatore di concatenazione di stringhe «+» effettua questa trasformazione in modo automatico.

A solo titolo di esempio sono riepilogati nelle TABELLE **4** e **5** alcuni dei metodi più utilizzati delle classi *wrapper* dei tipi primitivi.

TABELLA **4**

Classe	Metodo per acquisizione del valore incapsulato	Esempio di codice per la dichiarazione e la costruzione di un oggetto *wrapper* da valore di tipo primitivo
Integer	`int intValue()`	`Integer i = new Integer(123);`
Double	`double doubleValue()`	`Double d = new Double(3.1416);`
Boolean	`boolean booleanValue()`	`Boolean b = new Boolean(true);`
Character	`char charValue()`	`Character d = new Character('F');`

TABELLA **5**

Classe	Metodo per trasformazione da stringa di caratteri a valore numerico	Metodo per trasformazione da valore numerico a stringa di caratteri
Integer	`int parseInt(String s)`	`String toString(int value)`
Long	`long parseLong(String s)`	`String toString(long value)`
Float	`float parseFloat(String s)`	`String toString(float value)`
Double	`double parseDouble(String s)`	`String toString(double value)`

Tradizionalmente l'operazione di incapsulare un valore di tipo primitivo in un oggetto della corrispondente classe *wrapper* e, viceversa, l'operazione di estrarre il valore di un tipo di dato primitivo da un oggetto della corrispondente classe *wrapper* prendono rispettivamente il nome di ***boxing*** e ***unboxing*** (da *box*, «scatola»). Nelle più recenti versioni del linguaggio Java è stato introdotto l'***autoboxing***, cioè la conversione automatica di un tipo di dato primitivo nel corrispondente oggetto della classe *wrapper* e, viceversa, la conversione automatica di un oggetto di una classe *wrapper* nel corrispondente tipo di dato primitivo.

ESEMPIO

Con la funzionalità di *autoboxing* è possibile scrivere istruzioni come le seguenti:

```
…
Integer x = 123, z;
int y = x;
z = y;

…
```

OSSERVAZIONE La tecnica dell'*autoboxing* porta alla creazione di oggetti senza invocazione esplicita dell'operatore `new`.

Lo scopo principale delle classi è quello di istanziare oggetti, ma nel caso delle classi *wrapper* sono importanti anche i servizi di utilità forniti dai metodi statici che non richiedono di essere applicati a oggetti. Esistono classi

progettate esclusivamente per fornire servizi di utilità, prive di costruttori e con solo metodi statici: un esempio notevole è la classe *Math* del *package java.lang*, che non necessita di essere importato.
Dato che il linguaggio Java non prevede la dichiarazione di funzioni, ma esclusivamente di metodi di classe e che i valori numerici non sono oggetti, ma tipi primitivi, si ha la necessità di metodi per i calcoli delle funzioni matematiche (radici, logaritmi, esponenziali, funzioni trigonometriche, …).
Una possibile soluzione avrebbe potuto prevedere l'inserimento nella classi *wrapper Double e Float* di un metodo per ogni funzione matematica, ma questo approccio sarebbe stato poco pratico, in quanto avrebbe richiesto la creazione di un oggetto per il calcolo di una funzione; la soluzione seguita dai progettisti del linguaggio Java è stata quella di realizzare la classe *Math* come raccolta di metodi pubblici statici, ognuno dei quali implementa una specifica funzione matematica di cui nella TABELLA **6** si riportano le principali.

TABELLA **6**

Tipologia di funzioni matematiche	Funzioni della classe *Math*
Potenze e radici	`pow()`, `sqrt()`, …
Trigonometriche	`sin()`, `cos()`, `tan()`, …
Logaritmiche ed esponenziali	`log()`, `exp()`, …
Altre	`abs()`, `floor()`, `round()`, …

ESEMPIO

Il codice del metodo *main* della seguente classe esemplifica l'uso dei metodi della classe *Math*:

```
public class EsempioMath{
  public static void main(String args[]) {
    double x, y;
    x = 2.0;
    y = Math.sqrt(x);
    System.out.println("La radice quadrata di "+x+" e' "+y);
    x = Math.PI; // PI e' un attributo statico costante di Math
    y = Math.cos(x);
    System.out.println("Il coseno di "+x+" e' "+y);
  }
}
```

7 Stringhe e codifica Unicode

Il linguaggio Java non ha un tipo primitivo per le stringhe di caratteri, tuttavia rende disponibili una classe denominata *String* e varie funzionalità che ne permettono l'uso con facilità. Come in tutti i linguaggi che derivano la propria sintassi dal C/C++, una costante di tipo stringa è una sequenza di caratteri delimitata dal simbolo «"» come ad esempio "Hello, world!". Una stringa è quindi una sequenza di caratteri considerati come un unico elemento e un oggetto di tipo *String* assume come valore tale sequenza.

ESEMPIO

La seguente dichiarazione Java definisce l'oggetto *messaggio* cui viene successivamente assegnata la stringa «Hello, world!»:

```
String messaggio;
messaggio = "Hello, world!";
```

Lo stesso risultato si sarebbe ottenuto con la seguente riga di codice:

```
String 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 alti 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.output.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* e *s2* hanno lo stesso contenuto, **`false`** altrimenti; non è possibile utilizzare l'operatore «==» per effettuare questo confronto perché verificherebbe se quanto riferito da *s1* e *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 tutte i caratteri «A» sono stati sostituiti con caratteri «a».

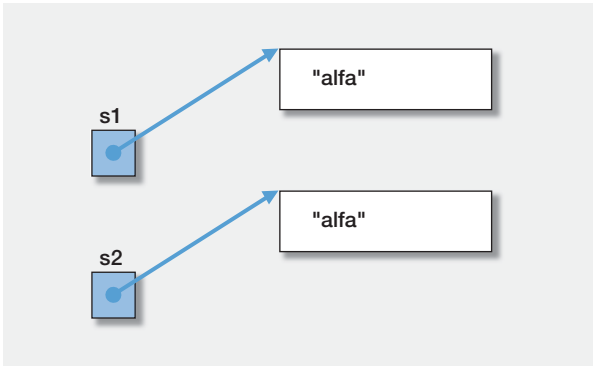


FIGURA **8**

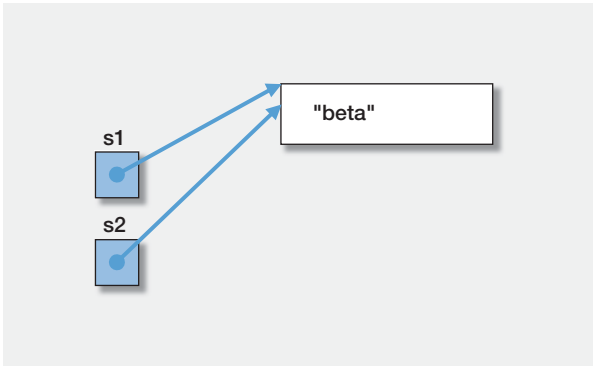


FIGURA **9**

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**).
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* e *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:

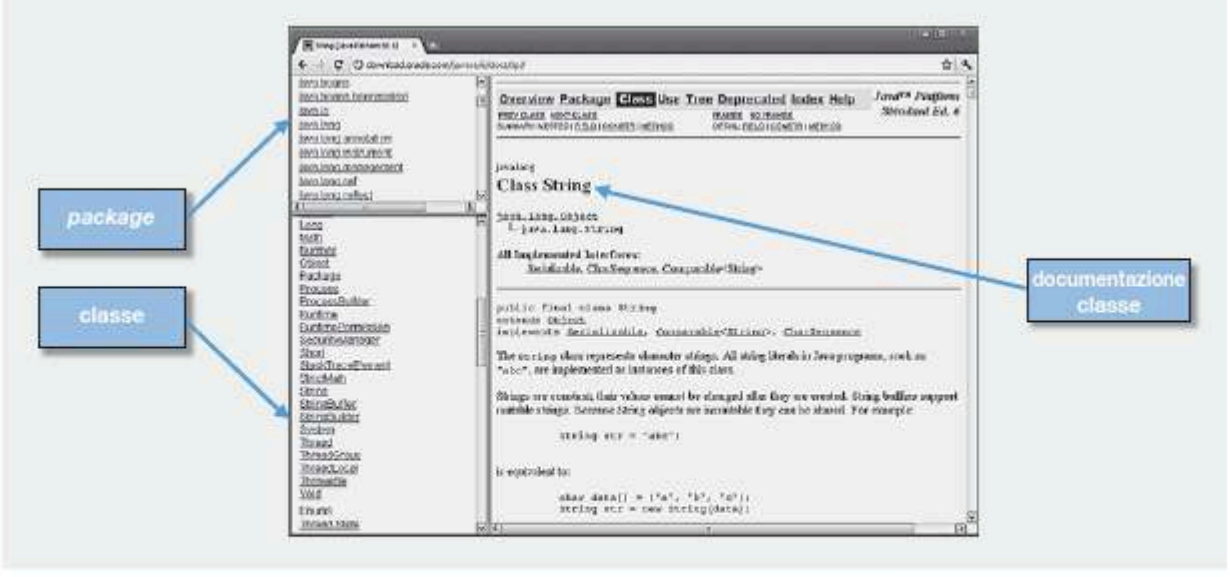
```
String s = "alfa";
StringBuffer sb = new StringBuffer(s);
sb.setCharAt(2,'b');
s = sb.toString();
```

8 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

FIGURA **10**



documentazione della libreria standard del linguaggio disponibile in rete viene creata con questo strumento.
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 diverrà parte della documentazione HTML. La TABELLA **8** riporta i *tag Javadoc* fondamentali.

TABELLA **8**

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 una 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, luminosita'=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
   */
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
   */
private void setColore(String c) {colore = c;}
/**
   * restituisce la tipologia di schermo del televisore
   */
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
   */
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
   */
```

```
public int getVolume() {return volume;}
/**
  * restituisce il livello di luminosita' impostato
  */
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 alzaVolume() {if (volume<50) volume++;}
/**
  * deremenda il livello di volume
  */
public void abbassaVolume() {
  if (volume>0) volume--;
}
/**
  * incrementa il livello di luminosità
  */
  public void aumentaLuminosita() {
    if (luminosita<80) luminosita ++;
  }
  /**
    * decrementa il livello di luminosità
    */
  public void diminuisceLuminosita() {
    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, rispettivamente per la parte relativa alla classe e per la parte relativa al costruttore e ai metodi.

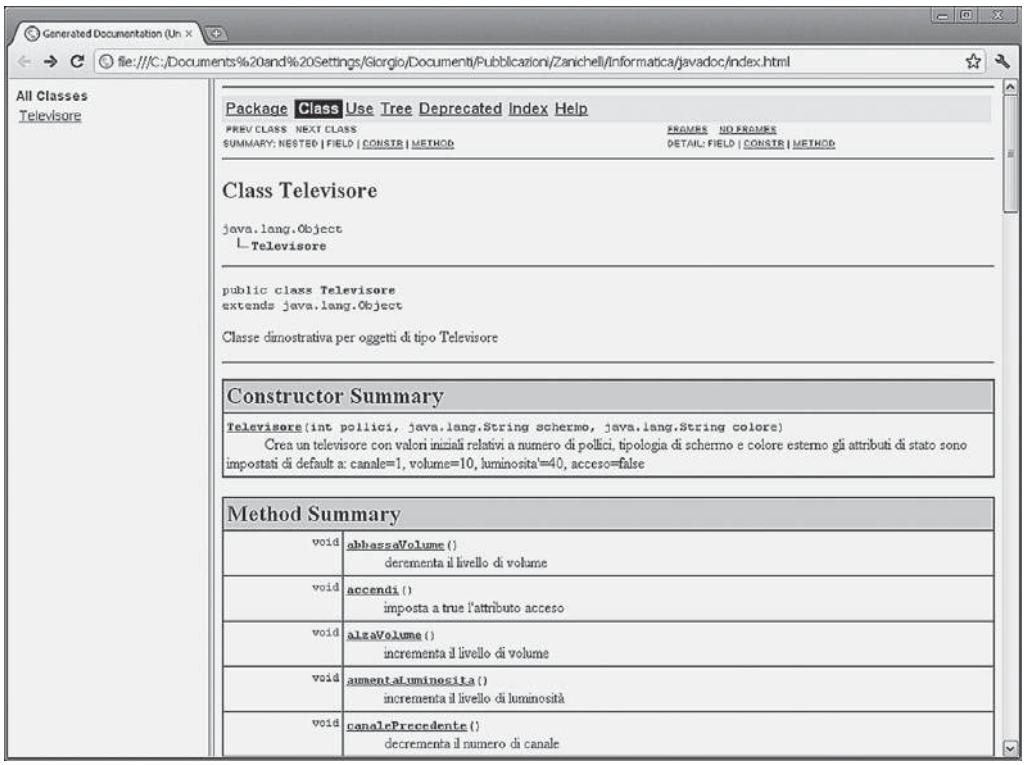


FIGURA **11**

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.

Sintesi

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

Bytecode. Il compilatore Java non produce direttamente codice eseguibile, ma un codice intermedio, *bytecode*, che viene eseguito dall'interprete Java.

JVM (***Java Vitual Machine***). È l'interprete che esegue il *bytecode* dei programmi Java. Ogni piattaforma hardware/software ha una sua propria specifica JVM per cui il *bytecode* 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 *bytecode*, 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 una 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 una 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 risultino accessibili e modificabili 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.

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.

QUESITI

1 **Il *bytecode* 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 Nessuna delle risposte precedenti.

2 **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 *bytecode*.

D La JVM è un componente hardware.

3 **La portabilità del codice Java è data dal fatto che …**

A … il *bytecode* 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 … che la JVM sia sempre la stessa su qualsiasi piattaforma hardware/software.

D Nessuna delle risposte precedenti.

4 **Il motto «*Compile once*, run *anywhere*» significa che …**

A … il *bytecode* prodotto dalla compilazione Java può essere utilizzato su qualunque piattaforma identica alla piattaforma di compilazione indipendentemente dalla JVM.

B … il *bytecode* prodotto dalla compilazione Java può essere utilizzato su qualunque piattaforma hardware/software solo se fornita di JVM.

C … il *bytecode* prodotto dalla compilazione Java può essere utilizzato su qualunque piattaforma hardware solo se ha lo stesso sistema operativo della piattaforma di compilazione e se fornita di JVM.

D … il *bytecode* prodotto dalla compilazione Java può essere utilizzato su qualunque piattaforma software solo se ha lo stesso hardware della piattaforma di compilazione e se fornita di JVM.

5 **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 Nessuna delle risposte precedenti.

6 **Due oggetti diversi istanza della stessa classe …**

A … condividono solo il valore degli attributi.

B … condividono solo i metodi.

C … condividono metodi e struttura generale.

D … condividono codice dei metodi e tipo degli attributi.

7 **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.

8 **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 Nessuna delle risposte precedenti.

9 Quale è la funzione dell'operatore Java `new`?

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

10 È possibile creare due riferimenti distinti a uno stesso oggetto?

A Sì, sempre.
B No, mai.
C Sì, ma solo 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.

11 Nel linguaggio Java lo spazio di memoria assegnato agli attributi di un oggetto istanziato 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 Nessuna delle risposte precedenti.

12 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)`

13 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

14 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 Il primo metodo che viene applicato ogni volta che si istanzia un oggetto.
C È un metodo statico.
D Non prevede parametri.

15 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.

16 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;*.

17 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*.

18 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 una notazione del tipo `NomeClasse.nomeMembro`.

D Nessuna delle risposte precedenti.

19 **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.

20 **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`

21 **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*.

22 ***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 Nessuna delle risposte precedenti.

23 ***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.

ESERCIZI

1 **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.**

2 **Si devono gestire degli oggetti di tipo *Angolo* nella forma g°m′s″ dove *g* rappresenta i gradi, *p* i primi e *s* i secondi (con $0 \le g < 360$, $0 \le p < 60$, $0 \le s < 60$). Dopo avere 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.

3 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.

4 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.

5 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.

6 Una 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.

7 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

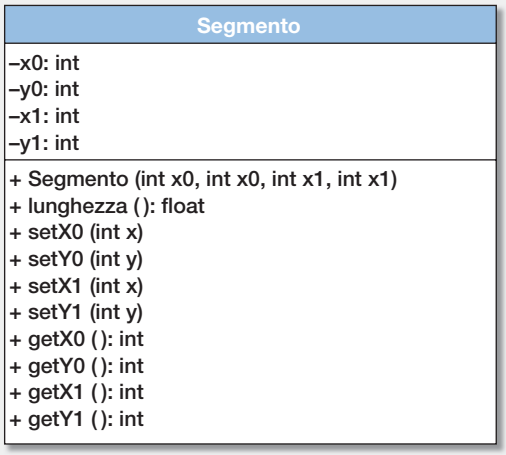


FIGURA **13**

della Terra individuato dalle proprie coordinate geografiche (latitudine e longitudine espresse in gradi, primi e secondi).

8 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**.

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.

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

10 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.

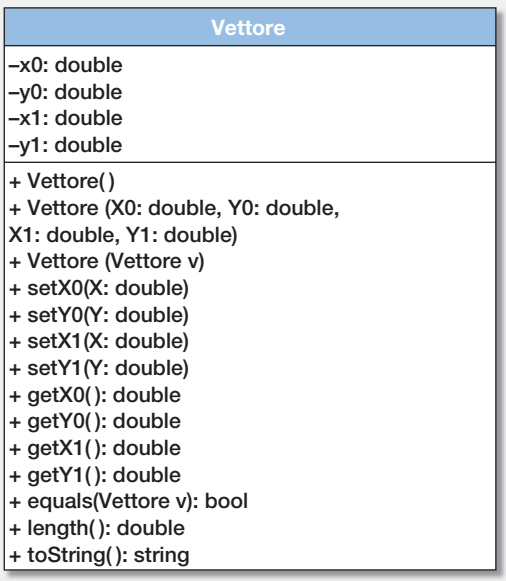


FIGURA **14**

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 `string2`, 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));
```

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

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

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.

span
distanza, intervallo.

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 dovere modellare una mensola sopra la quale collocare dei libri. Un possibile diagramma UML potrebbe essere quello di 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;                                          ▶
```

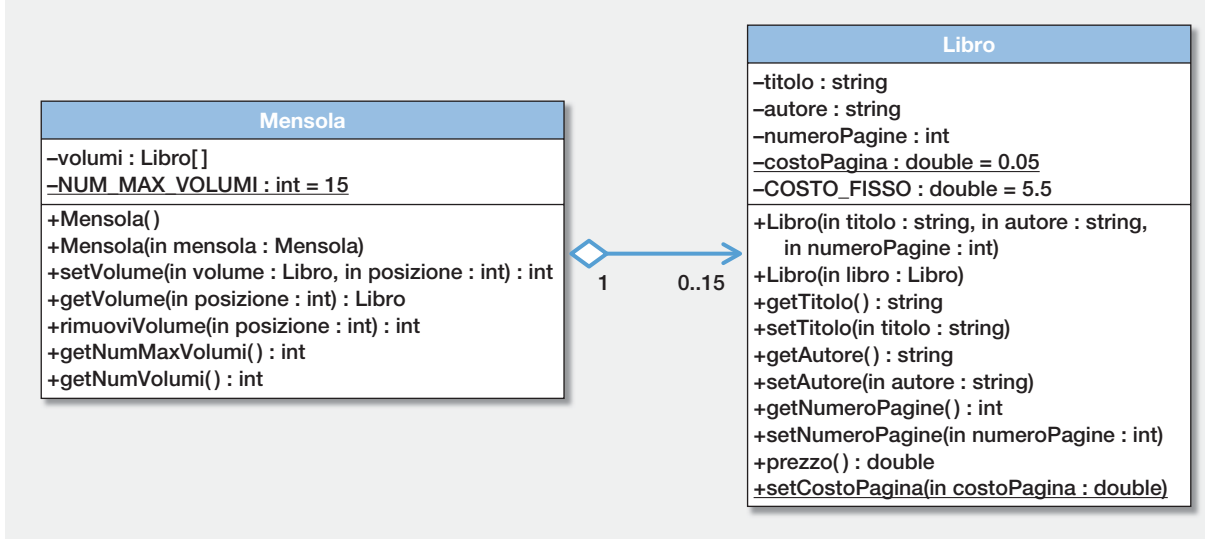


FIGURA **1**

```
    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;
  }
}
```

OSSERVAZIONI

- 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[];
```

OSSERVAZIONE 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.

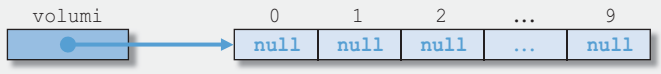
Nel caso di dichiarazione di *array* di oggetti di una determinata classe, i singoli elementi dell'*array* contengono i **riferimenti** agli oggetti.

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**.

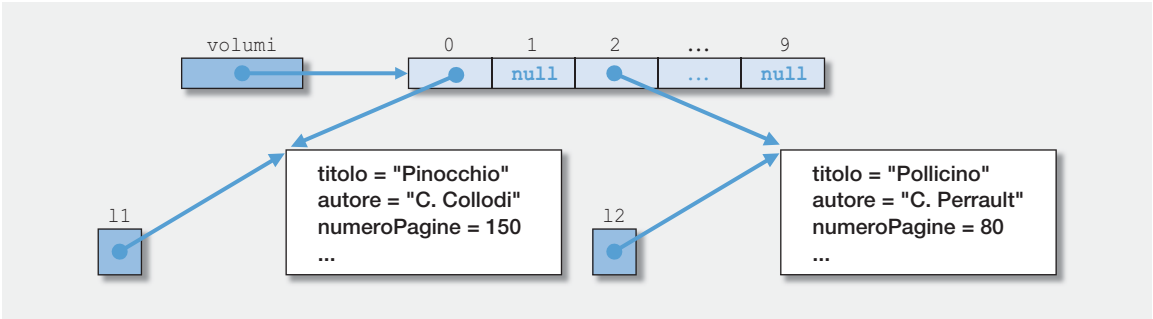


FIGURA **2**

Se successivamente viene eseguita la seguente istruzione:

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

si avrà la situazione di FIGURA **3**.

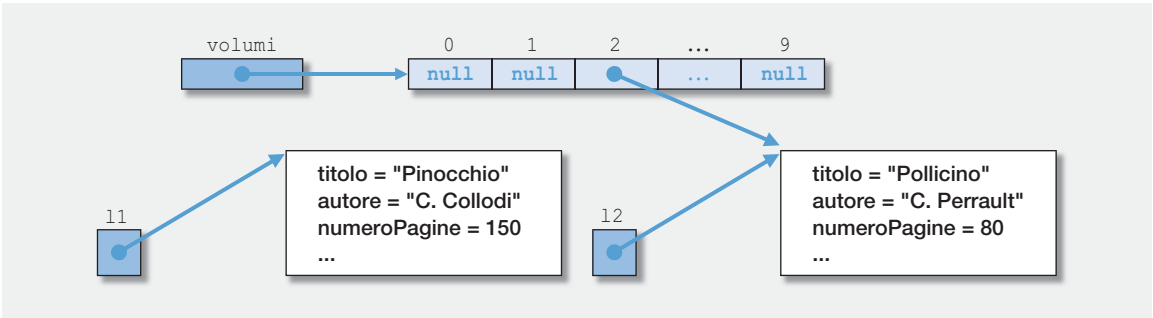


FIGURA **3**

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**.

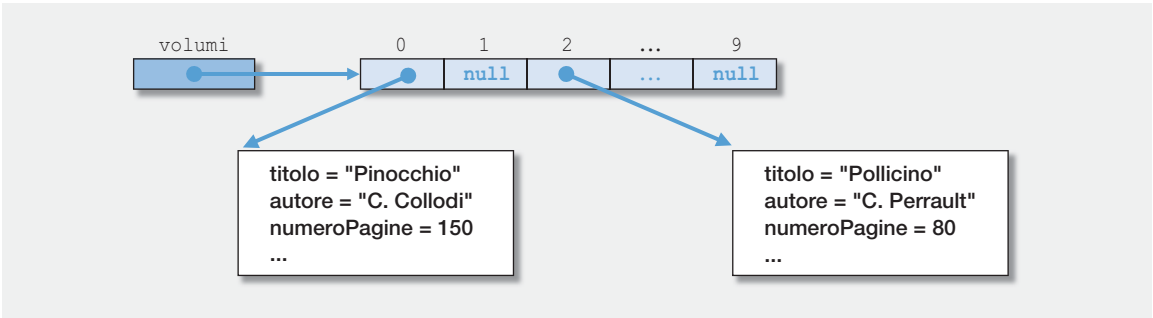


FIGURA **4**

Quindi, dopo l'esecuzione dell'istruzione

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

sarebbe divenuta 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.

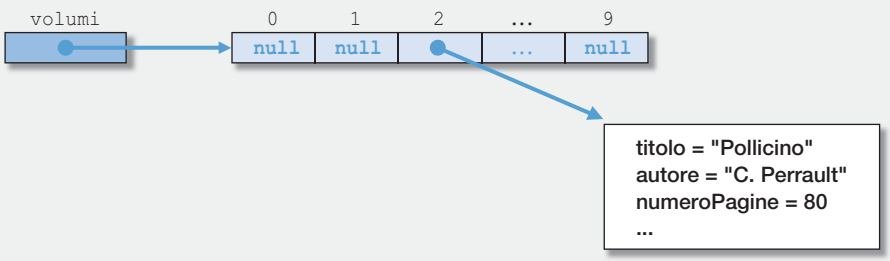


FIGURA **5**

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 che multidimensionale, esse adottano la stessa sintassi di altri linguaggi di programmazione, come ad 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 estremamente complesse.

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++) {
  …
  …
  …
}
```

OSSERVAZIONE 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, come 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.getNumLibri(ripiano);
         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.getNumLibri(ripiano);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: 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: 1 posizione: 0 -> La bella addormentata nel bosco 8.0€
ripiano: 1 posizione: 1 -> Cappuccetto Rosso 13.0€
```

OSSERVAZIONE 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 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* sono 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 nella FIGURA **7**.

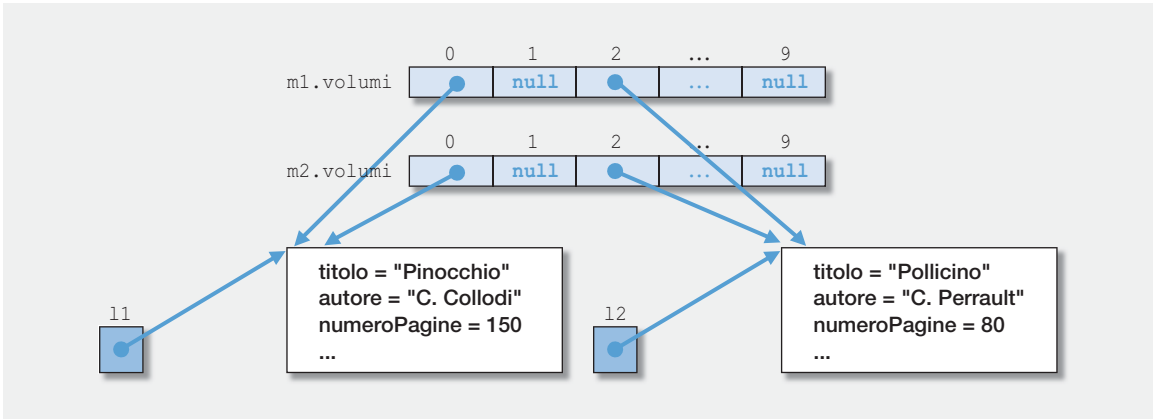


FIGURA **7**

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* e *m2* (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:

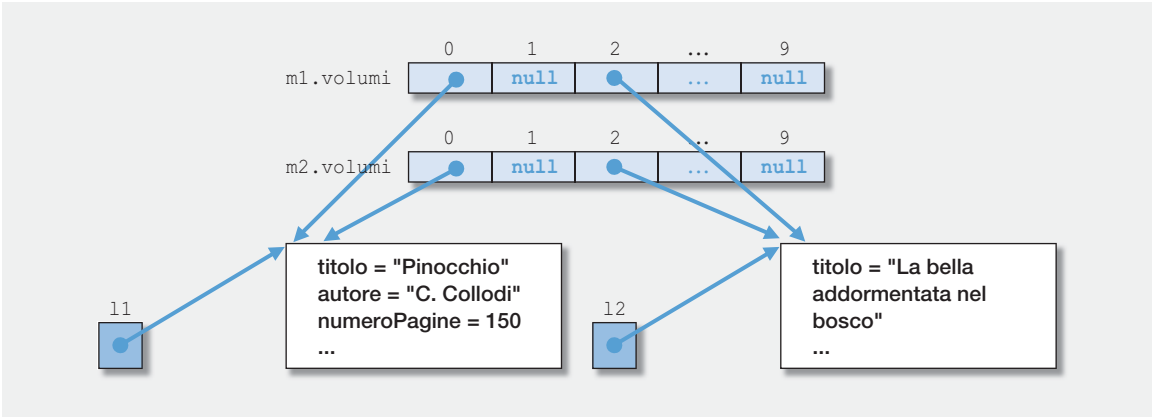


FIGURA **8**

```
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* e *m2* hanno ora copie separate dei propri attributi ed eventuali modifiche apportate a uno dei due oggetti sono ininfluenti per l'altro.

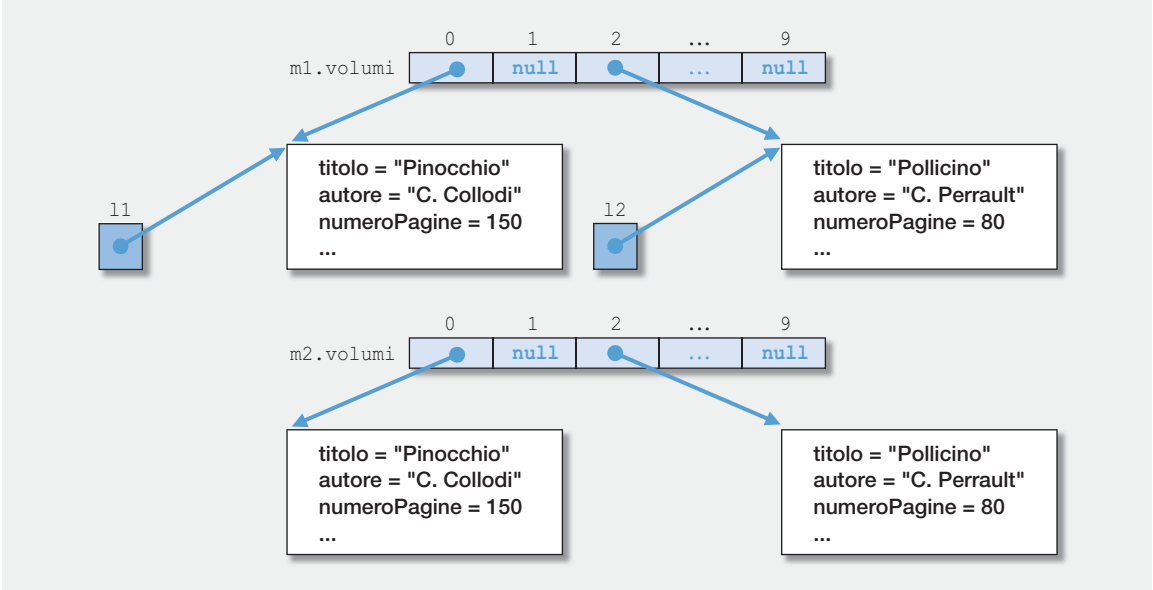


FIGURA **9**

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 che di ottenerlo come valore di ritorno.

ESEMPIO

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

- 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];
  libriAutore = 0;
  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();
        libriAutore++;
      }
  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 produce 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 nuovaMensola = new Mensola(elencoLibri);
```

Se nel *main* inseriamo 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);

// 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

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* del 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.

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 del 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;
- **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.

OSSERVAZIONE 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 (ad 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 della 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; …

OSSERVAZIONE 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.

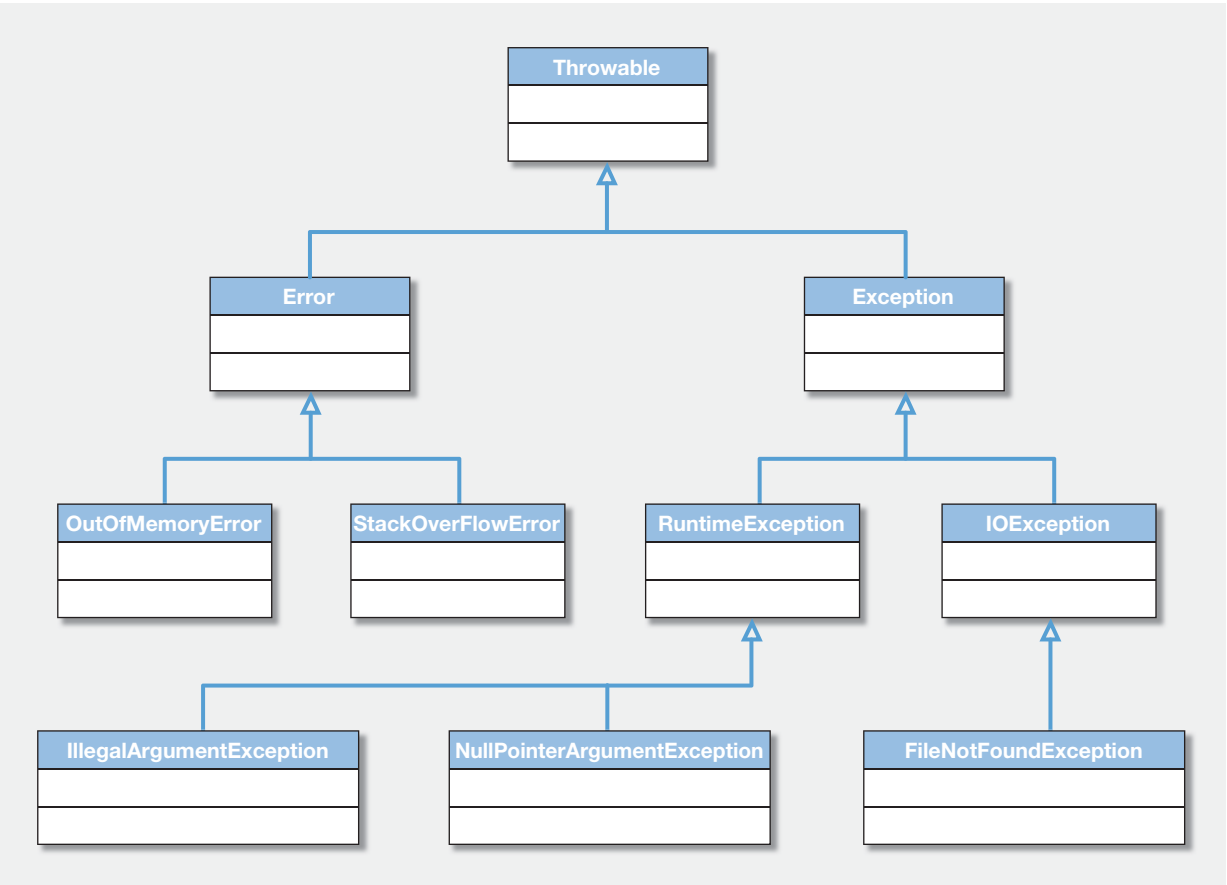


FIGURA **10**

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**.

OSSERVAZIONE Le eccezioni che derivano dalla classe *Error* (tipicamente relative al funzionamento della JVM), pur essendo intercettabili, sono in ogni caso irrecuperabili. Le eccezioni che derivano dalla classe *RuntimeException* sono eccezioni *unchecked* («non controllate»), che il programmatore può decidere di intercettare o meno; le altre eccezioni che derivano dalla classe *Exception* sono di tipo *checked* 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;
  }
}
```

OSSERVAZIONE 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 una 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*:

```
/* 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) {
  if (h>=0 && h<24 && m>=0 && m<60 && s>=0 && s<60)
    return true;
  else
    return false;
}

/* 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.

OSSERVAZIONE 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.

OSSERVAZIONE Le eccezioni dichiarate dal programmatore in linguaggio Java sono normalmente di tipo *checked* («controllato»), 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, segnala 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.

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(Lib  ro libro, int ripiano, int posizione)
            throws PosizioneNonValida, PosizioneNonVuota {
  int err;
  try {
    if ((err = ripiani[ripiano].setVolume(libro, posizione)<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) {
    System.out.println(exception);
  }
  catch (PosizioneNonVuota exception) {
    System.out.println(exception);
  }
  try {
    scaffale.setLibro(l3, 0, 10);
  }
```

```
catch (PosizioneNonValida exception) {
  System.out.println(exception);
}
catch (PosizioneNonVuota exception) {
  System.out.println(exception);
}

// inserimento mensola #1
try {
  scaffale.setLibro(l3, 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);
      }
    }
  }
```

che 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.

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:

```
InpuStreamReader 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 = keybaord.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 altri 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 convertire 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, spesso 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.

OSSERVAZIONE 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.

Sintesi

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 St5ring[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 varizione dell'uno si rifletterebbe in una automatica e spesso inaspetatta 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 basa 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*, *NullPointerExceptin*, *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, spesso 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*.

QUESITI

1 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 Nessuna delle risposte precedenti.

2 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.

3 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 **?????**.
D … si dichiara un vettore di riferimenti a oggetti di tipo `int`.

4 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 Nessuna delle risposte precedenti.

5 L'attibuto *length* di un *array* contiene …

A … il numero di elementi dell'*array*.
B … il numero di elementi non vuoti dell'*array*.
C … il numeto di elementi vuoti dell'*array*.
D … il numero di byte occupato dall'*array* nell'area di memoria *heap*.

6 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 Nessuna delle risposte precedenti.

7 **Le gestione delle eccezioni avviene in Java tramite il costrutto …**

A … `if`/`else`.
B … `try`/`catch`.
C … `switch`/`case`.
D … `do`/`while`.

8 **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`.

9 **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.

10 **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*.

11 **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 Nessuna delle risposte precedenti.

12 **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 a fianco, 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.

13 **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 Nessuna delle risposte precedenti.

14 **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.

```
…
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;
    }
}
…
```

ESERCIZI

1 Facendo riferimento alla classe *Programma* progettata e implementata nell'esercizio 3 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 avere 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.

2 Facendo riferimento alla classe *CD* progettata e implementata nell'esercizio 4 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 avere 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.

3 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.

4 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.

5 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.

6 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*).

7 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 5 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 dato il codice o la targa;
- ricerca delle informazioni di un veicolo dato 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.

8 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.

9 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.

10 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 avere 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 6 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.

LABORATORIO

1 Si vuole simulare un cronometro che effettua 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.

Dotare inoltre la classe di un costruttore che azzera 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:

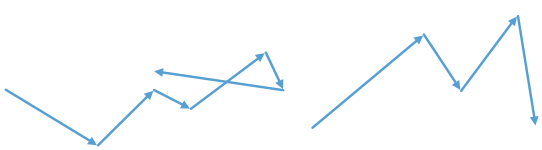
- *start* di *c1*
- *attesa 5 secondi*
- *show* in secondi del valore di *c1*
- *start* di *c2*
- *attesa 61 secondi*
- *stop* di *c1*
- *show* in minuti:secondi di *c1*
- *attesa 4 secondi*
- *start* di *c1*
- *attesa 2 secondi*
- *stop* di *c2*
- *show* di *c2* in secondi
- *stop* di *c1*
- *show* 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) {
   }
}
```

2 Facendo riferimento alla classe *Vettore* implementata nell'esercizio 10 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 a fianco.

Nei sistemi CAD (*Computer Aided Design*) e GIS (*Geogrphic Informatin Systems*) 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.

3 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

```
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]

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 nella pagina seguente.

```
#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;
}
```

Strutture dati

A4

Si deve realizzare un programma per la gestione dell'elenco degli invitati a una festa dove il numero di invitati 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 un struttura dati di questo tipo può essere rappresentata come in FIGURA **1**, dove:

- *head* («testa») è il riferimento di 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.

Per inserire, ad 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*.

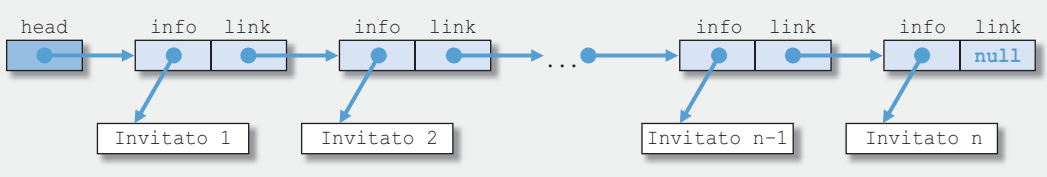


FIGURA **1**

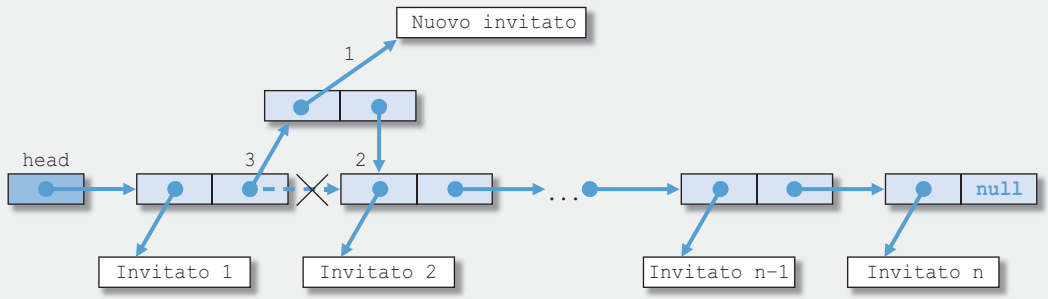


FIGURA **2**

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*).

Strutture dati come quella appena descritta prendono il nome di **liste**.
Il diagramma delle classi UML della lista degli invitati potrebbe essere quella mostrata in FIGURA **3**.

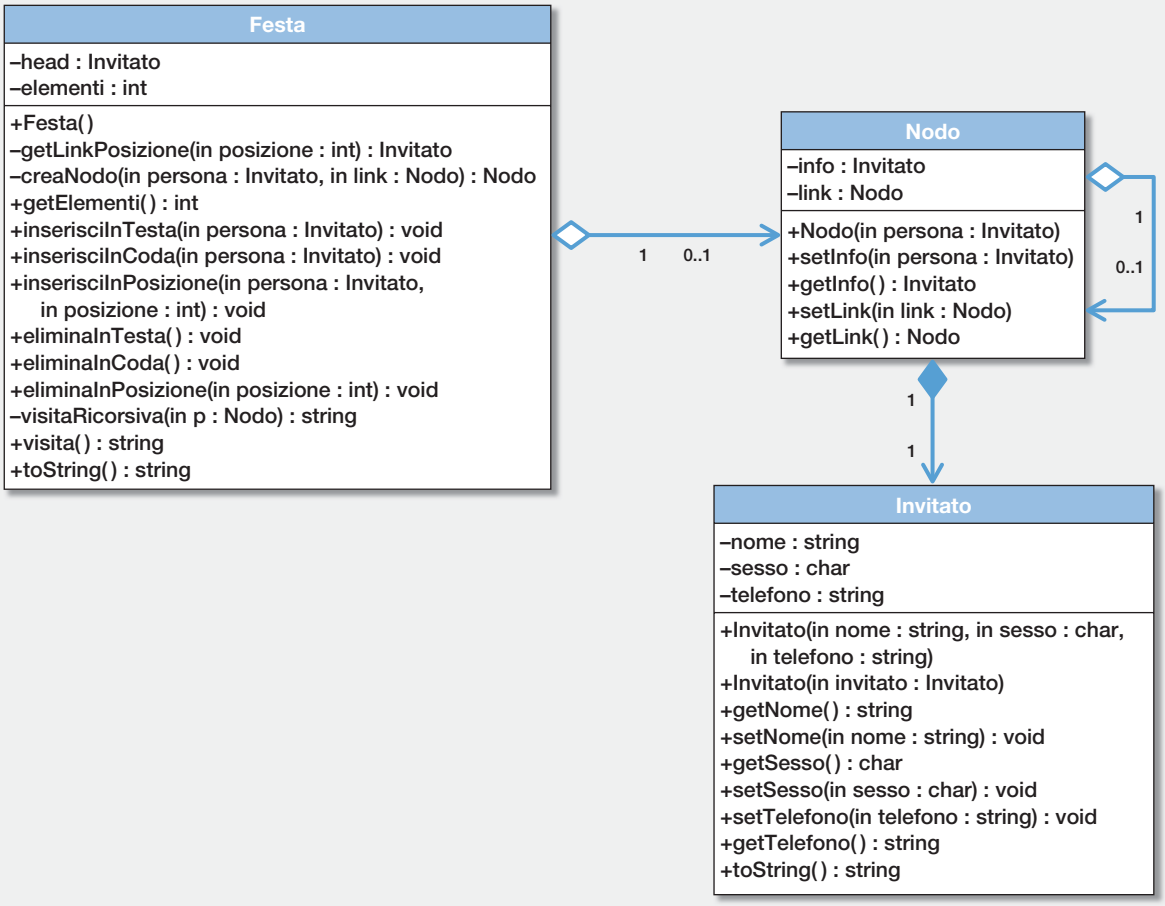


FIGURA **3**

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 che fisicamente possono occupare posizioni di memoria qualsiasi senza avere alcun ordinamento precostituito.

Nella TABELLA 1 abbiamo 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.

OSSERVAZIONE 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.

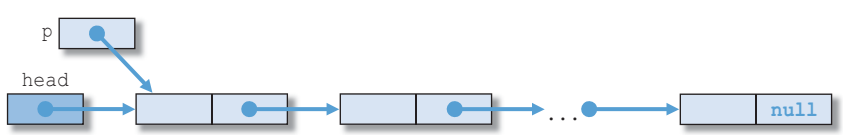
VISITA DEGLI ELEMENTI DI UNA LISTA

L'algoritmo che permette di scorrere tutti gli elementi di una lista può essere implementato con il metodo

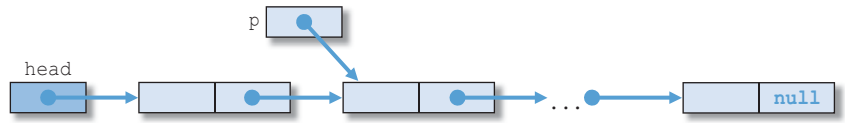
```
public void visitaLista() {
  Nodo p = head;

  while(p != null) {
    esamina(p.getInfo());
    p = p.getLink();
}
```

dove:

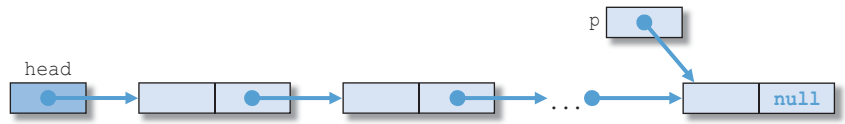
- la prima istruzione assegna alla variabile *p* utilizzata per scorrere gli elementi della lista il valore del riferimento iniziale della lista (*head*):



- 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.

OSSERVAZIONE 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:

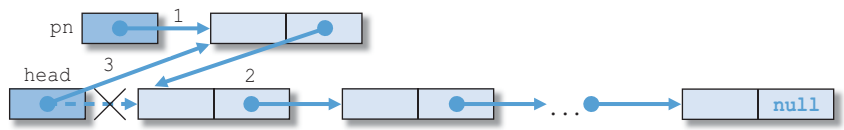
```
public void visitaLista() {
  visitaRicorsivaLista(head);
}
```

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.

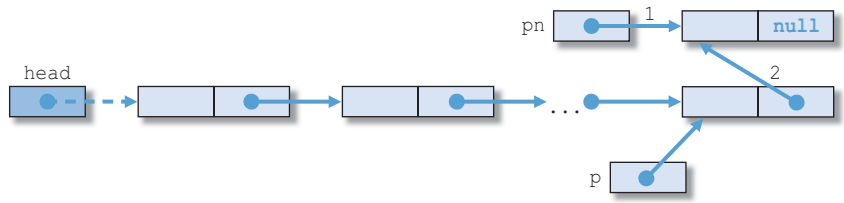
Nel caso di **inserimento in testa** di un nuovo nodo l'algoritmo è piuttosto semplice:

```
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 avere creato il nuovo elemento, scorrere la lista posizionandosi sull'ultimo nodo mediante una variabile di riferimento e collegare a esso il nuovo nodo:

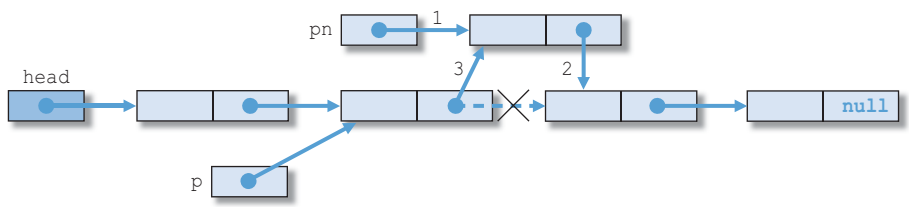
```
public void inserisciInCoda() {
  Nodo p = head; // riferimento per scorrere la lista
  Nodo pn = new Nodo(); // creazione nuovo nodo

  if (p==null) // lista vuota?
    inserisciInTesta();
  else {
    while (p.getLink()!=null) // scorrimento lista
      p = p.getLink();

    pn.setLink(null); // il nuovo nodo e' l'ultimo della lista
    p.setLink(pn);    // pn diviene il nuovo ultimo nodo
    elementi++;       // aggiornamento del numero di nodi
  }
}
```



OSSERVAZIONE 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();
      pn.setLink(p.getLink()); // pn riferisce il nodo successivo
      p.setLink(pn);           // il nuovo nodo viene collegato
      elementi++;              // aggiornamento del numero di nodi
    }
}
```



OSSERVAZIONE 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--;            // aggiorna del numero dei nodi
}
```

OSSERVAZIONE 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 qualsiasi operazione di eliminazione. Infatti, se la lista è vuota, l'istruzione `head = head.getLink()` genera un'eccezione perché la componente *link* non esiste: 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*:

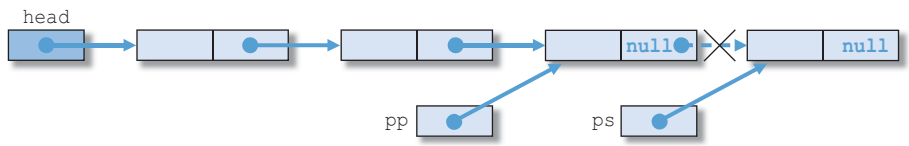
```
public void eliminaInCoda() {
  Nodo ps=head, pp=head; // variabili per scorrere la lista

  if (pp==null) // lista vuota?
    return;
  while(ps.getLink() != null) { // scorrimento della lista
    pp = ps;
    ps = ps.getLink();
  }
```

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.

```
  if (pp==head) // solo un nodo nella lista?
    eliminaInTesta();
  else {
    pp.setLink(null);  // il penultimo nodo diviene ultimo
    elementi--;        // aggiornamento del numero dei nodi
  }
}
```



OSSERVAZIONE 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.

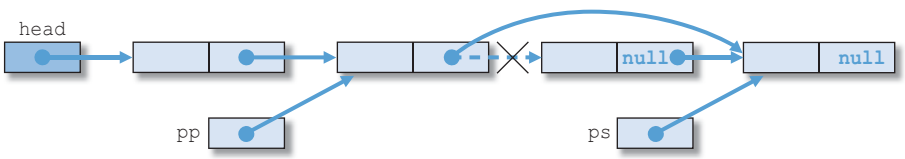
In alternativa si potrebbe utilizzare un metodo che, basandosi sul valore dell'attributo *elementi* (numero di nodi presenti nella lista) restituisca il riferimento di un nodo in posizione *n*-esima, oppure `null` se la posizione non è corretta:

```
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 una **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 i = 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 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.visita());
    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
```

OSSERVAZIONE 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

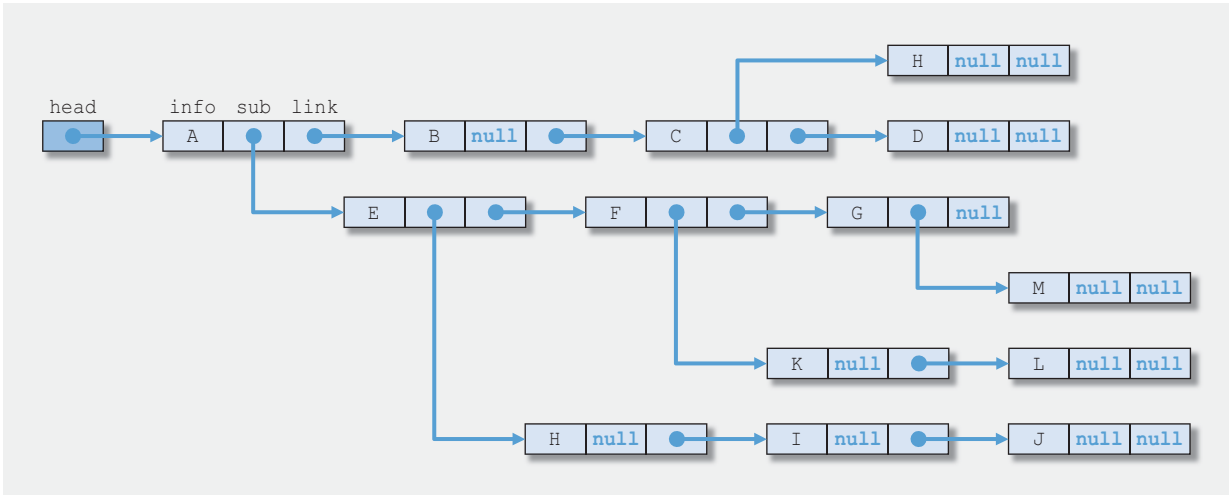


FIGURA 4

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.

OSSERVAZIONE 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*, [*H*], *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 patterns*** sono soluzioni generiche di progettazione software applicabili a problemi ricorrenti: questo approccio si è diffuso a partire dalla pubblicazione nel 1995 di *Design pattern: elementi per il riuso del software ad oggetti* di Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, da allora noti come la «Banda dei quattro» (*The «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

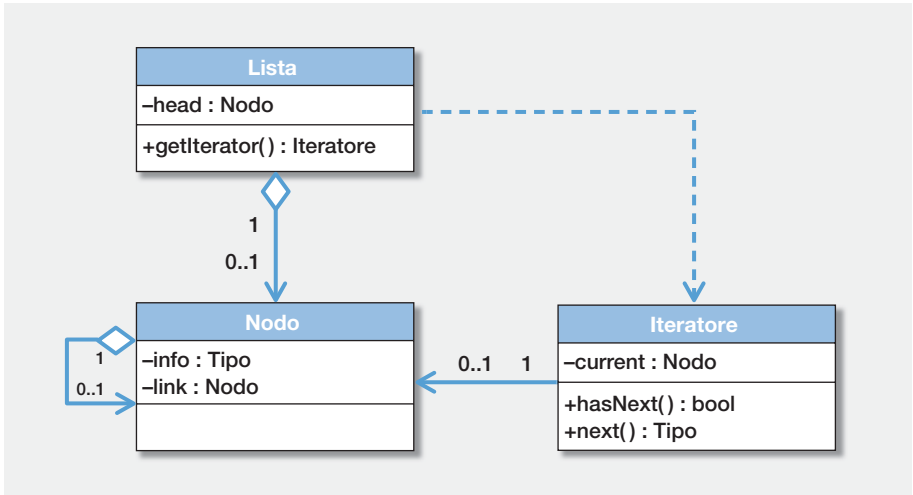


FIGURA **5**

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 linguaggio 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.

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);
}
…
```

che 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 La pila e la coda

Due strutture dati molto usate nella pratica sono la **pila** (*stack*) e la **coda** (*queue*).

3.1 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 nella FIGURA **6**, dove sono riportate le operazioni di inserimento e di estrazione di un elemento.
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.

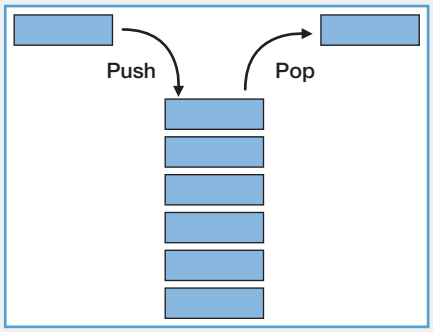


FIGURA **6**

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);

    Invitato inv = pila.pop();
    if (inv==null)
      System.out.println("Pila vuota");
    else
      System.out.println(inv);
    System.out.println(pila);
    inv = pila.pop();
    if (inv==null)
      System.out.println("Pila vuota");
    else
      System.out.println(inv);
    System.out.println(pila);
    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
```

3.2 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 l'ultimo a essere estratto.

ES. 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 nella FIGURA **7**, dove sono riportate le operazioni di inserimento e di estrazione di un elemento.

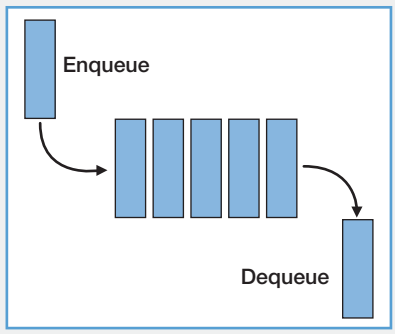


FIGURA **7**

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.

OSSERVAZIONE 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);
    Invitato inv = coda.dequeue();
    if (inv==null)
      System.out.println("Coda vuota");
    else
      System.out.println(inv);
    System.out.println(coda);
    inv = coda.dequeue();
    if (inv==null)
      System.out.println("Coda vuota");
    else
      System.out.println(inv);
    System.out.println(coda);
    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
```

4 Alberi

Gli **alberi** sono una struttura dati molto utilizzata in informatica perché permettono di memorizzare informazioni in modo dinamico, efficiente e ordinato. Un albero è una struttura gerarchica in cui ogni elemento, 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».

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**).

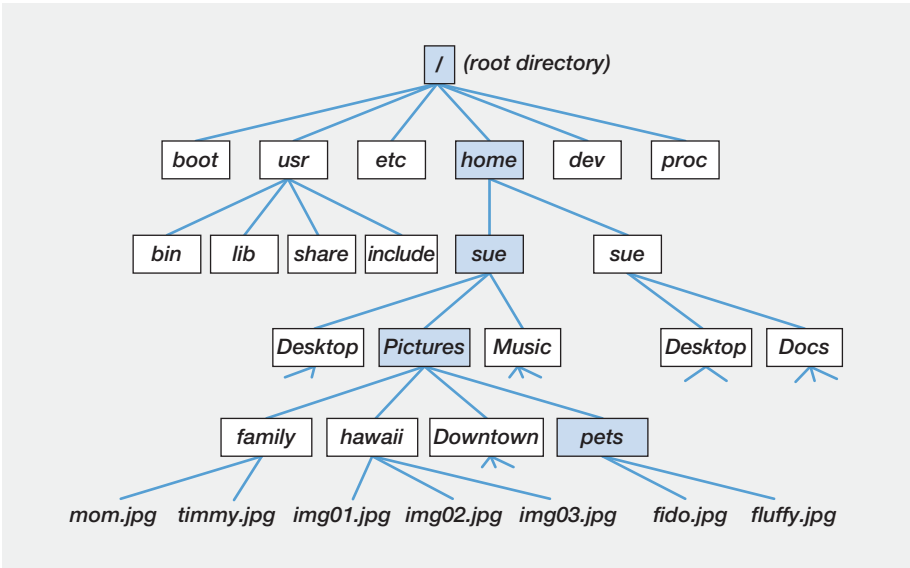


FIGURA **8**

4.1 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;
- sia **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*.

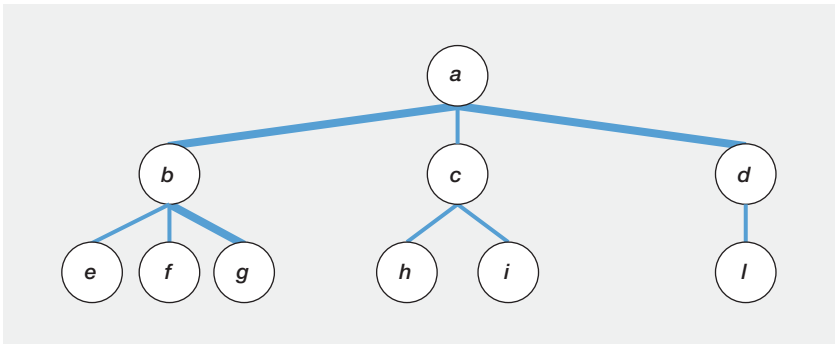


FIGURA **9**

Albero di gioco

I programmi di 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.

OSSERVAZIONE 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**.

OSSERVAZIONE 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*.

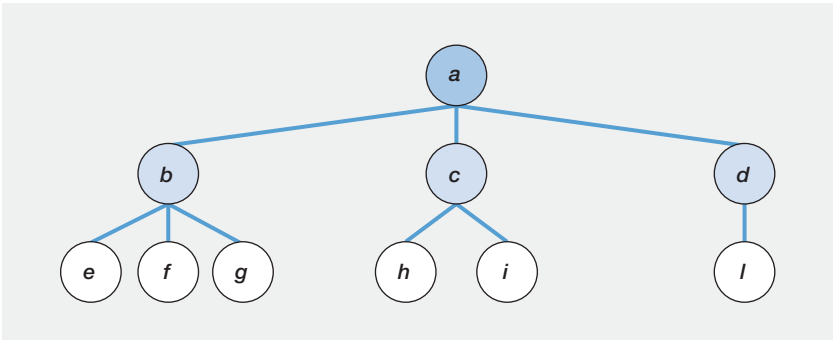


FIGURA **10**

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**.

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* e *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* e *l* è 2 (*e*, *f*, *g*, *h*, *i* e *l* appartengono al livello 2 dell'albero); di conseguenza la profondità massima dell'albero è 2.

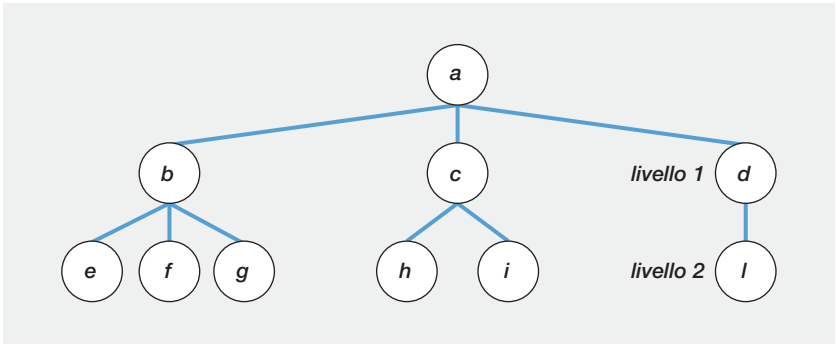


FIGURA **11**

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).

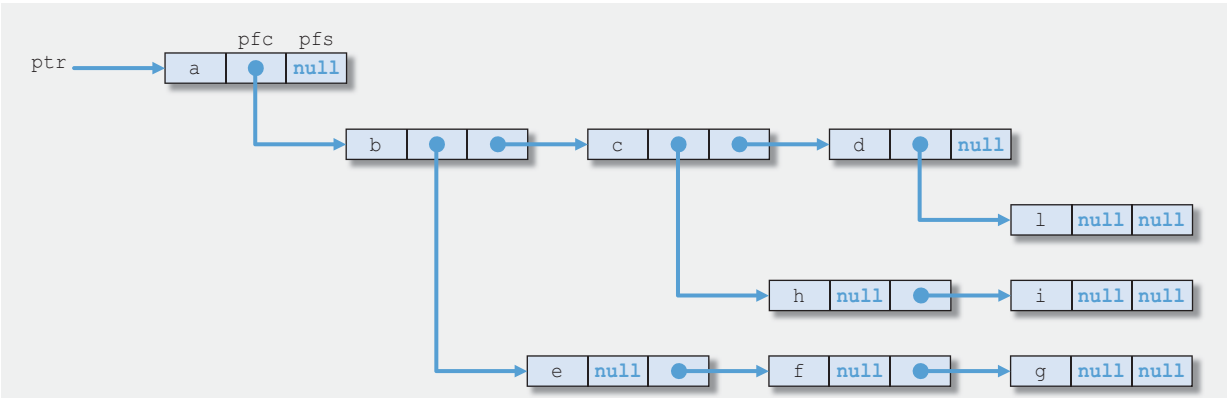


FIGURA **12**

OSSERVAZIONE 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**.

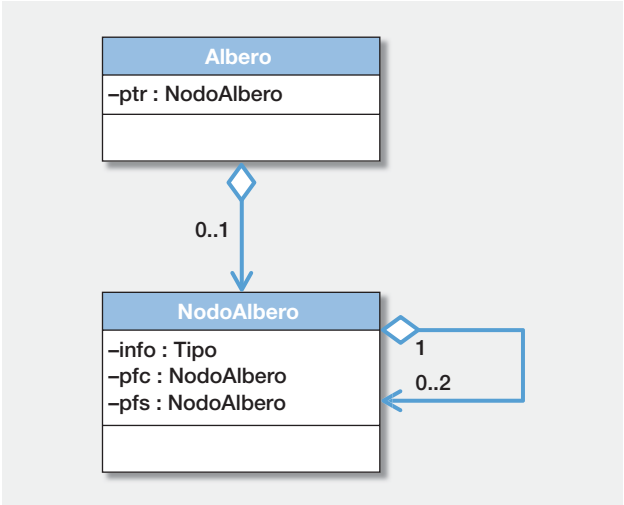


FIGURA **13**

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*.

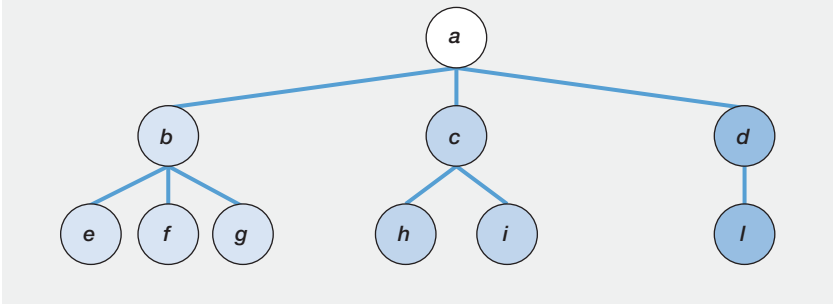


FIGURA **14**

OSSERVAZIONE 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);
}
```

ESEMPIO

Visitando l'albero di FIGURA **15** in modo differito si ottiene la sequenza *e*, *f*, *g*, *b*, *h*, *i*, *c*, *l*, *d*, *a*.

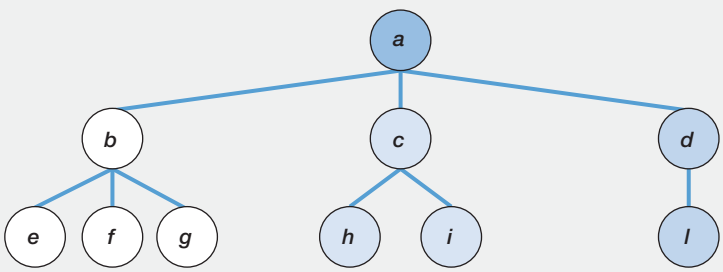


FIGURA **15**

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.

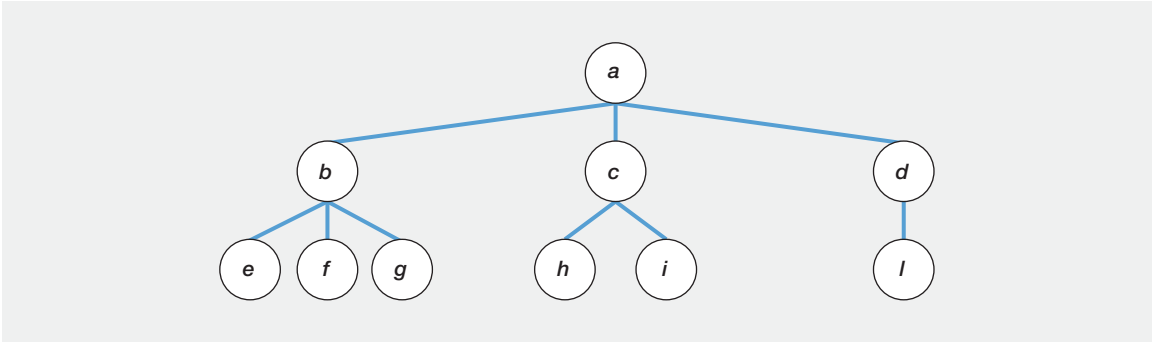


FIGURA **16**

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)
      visitaAnticipataRic(p.getFirstChild());
    if (p.getFirstSibling() != null)
      visitaAnticipataRic(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 aggiungiFratello(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) p.setFirstSibling(n);
  }

  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) p.setFirstChild(n);
  }

  public static void main(String[] args) {
    Albero a = new Albero();
    a.aggiungiFiglio('a', 'a');

    a.aggiungiFiglio('a', 'b');
    a.aggiungiFratello('b','c');
    a.aggiungiFratello('c','d');

    a.aggiungiFiglio('b', 'e');
    a.aggiungiFratello('e','f');
    a.aggiungiFratello('f','g');

    a.aggiungiFiglio('c', 'h');
    a.aggiungiFratello('h','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;

- i metodi *aggiungiFiglio* e *aggiungiFratello* prevedono entrambi 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 o l'utilizzo di più di un'istruzione *aggiungiFiglio* per un dato padre che, date le premesse e la modalità di memorizzazione dell'albero, invaliderebbero 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.

4.2 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:

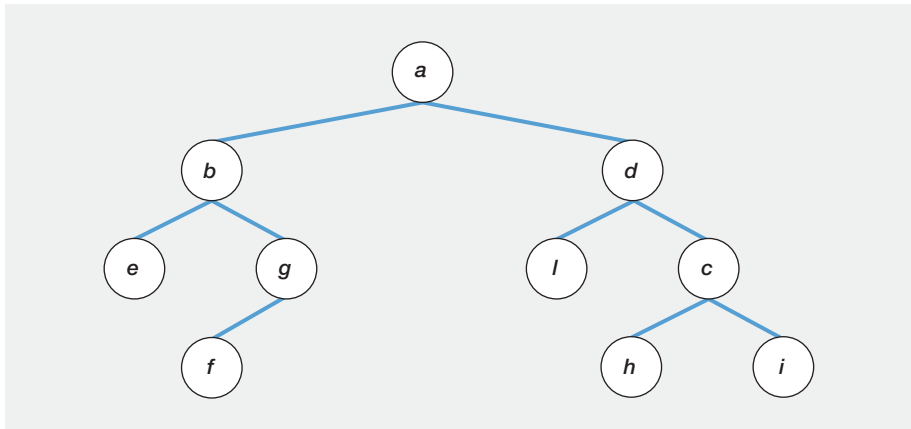


FIGURA **17**

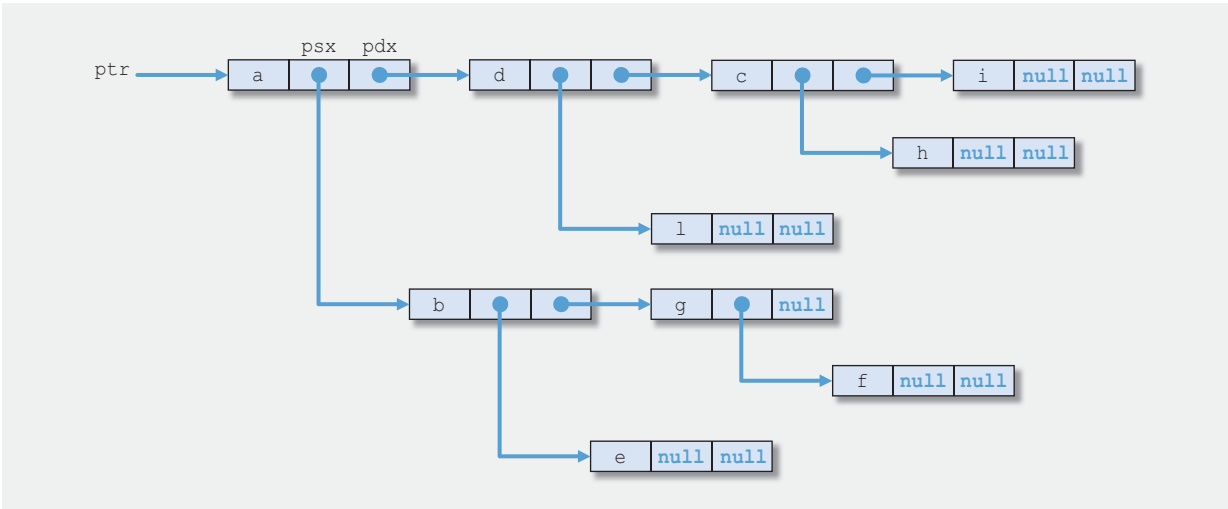


FIGURA **18**

- *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*).

ESEMPIO

Tralasciando la definizione delle operazioni, il diagramma UML delle classi che consentono di rappresentare un albero binario è come quello di FIGURA **19**.

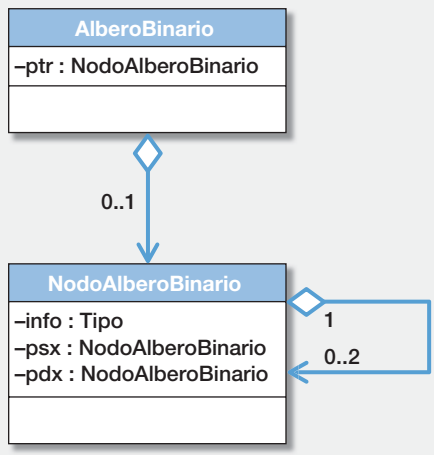


FIGURA **19**

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(NodoAlbero 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 NodoAlbero ptr;
  …
  …
  …
}
```

In un albero binario, oltre alle visite in ordine anticipato e differito, è possibile condurre una **visita in odine 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() {
    visitaAnticipata(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*.

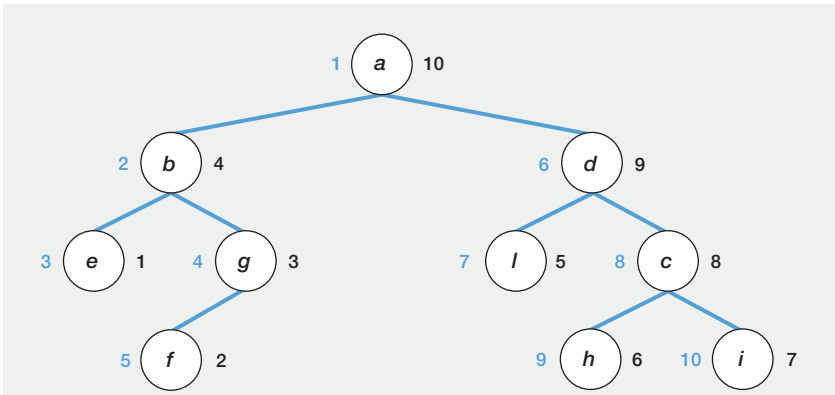


FIGURA **20**

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.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);
}
```

ES. 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*.

ES. Con la precedente trasformazione l'albero generico di FIGURA **21** diviene l'albero binario di FIGURA **22**.

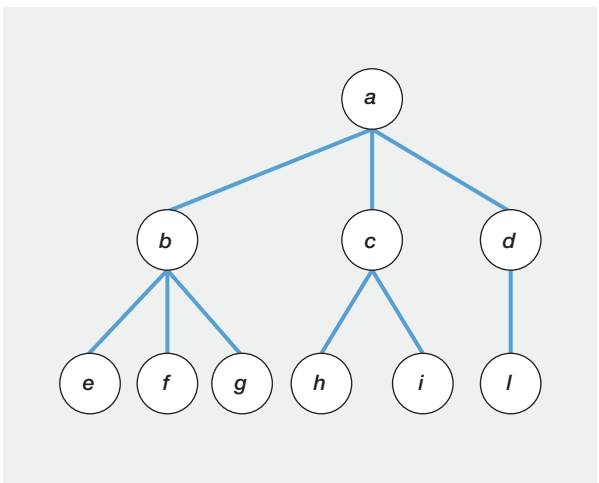


FIGURA **21**

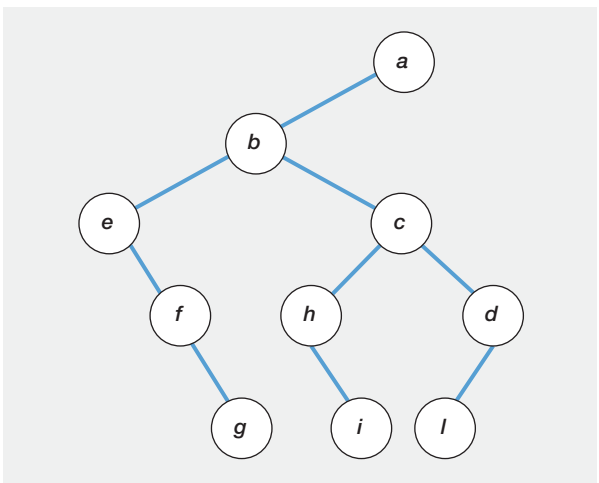


FIGURA **22**

OSSERVAZIONE 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.

4.3 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(NodoAlbero 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 NodoAlbero 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.

ESEMPIO

L'albero binario di FIGURA **23** è un albero binario di ricerca secondo il criterio dell'ordinamento alfabetico.

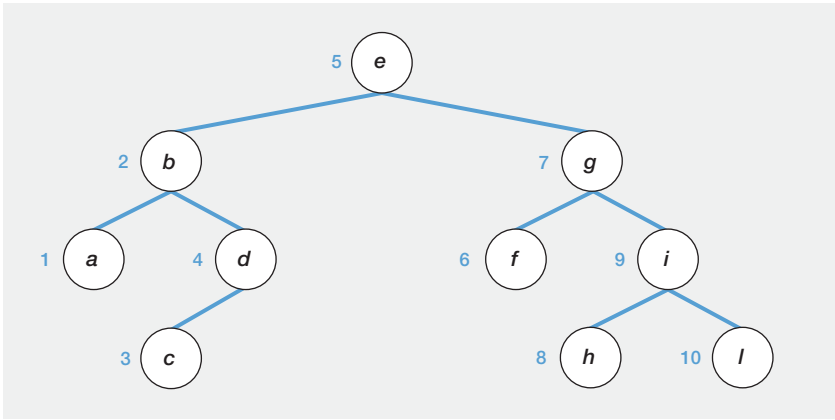


FIGURA **23**

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:

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).

```
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.

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 null;
}
```

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**).

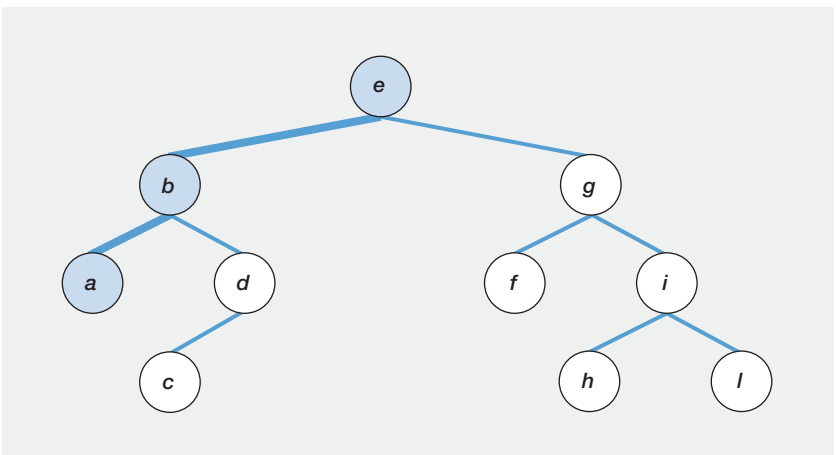


FIGURA **24**

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.

OSSERVAZIONE 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**).

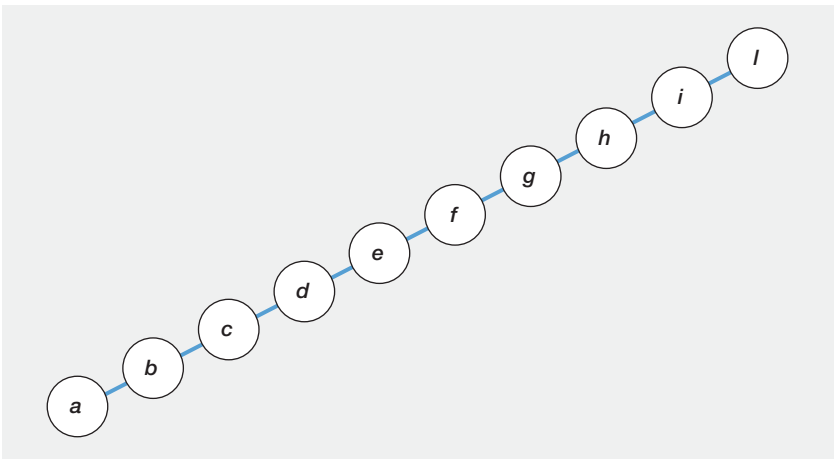


FIGURA **25**

5 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.

OSSERVAZIONE 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.

OSSERVAZIONE 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 al 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 $\log_2 N$.

OSSERVAZIONE 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.

OSSERVAZIONE

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].

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

$$h(k) = \sum_{i=0}^{n-1} chiave[i]$$

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.

2. Una particolare 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.

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

$$h(\text{"Rossi Mario"}) =$$
$$= 82 + 111 + 115 + 115 + 105 + 32 + 77 + 97 + 114 + 105 + 111 = 1064$$

OSSERVAZIONE 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.lenght;
  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

$$h(\text{"Rossi Mario"}) =$$
$$= (82 + 111 + 115 + 115 + 105 + 32 + 77 + 97 + 114 + 105 + 111) \bmod 100 = 64$$

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*;

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.

- 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.

OSSERVAZIONE 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.

OSSERVAZIONE

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.

5.1 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.

OSSERVAZIONE 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.

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.

▶ 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):

$$lf = \frac{n}{d}$$

OSSERVAZIONE 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 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 $k_1, k_2, \ldots$ per cui $h(k_1), h(k_2), \ldots$ 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 nella FIGURA **26**.

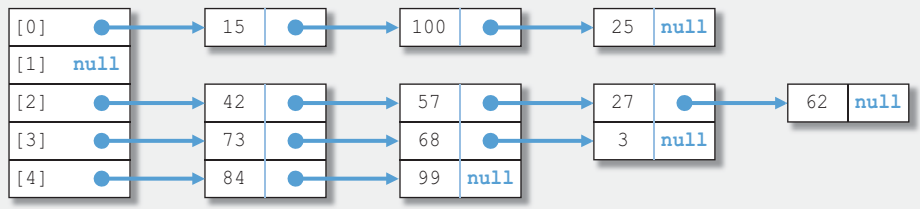


FIGURA **26**

OSSERVAZIONE 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.

OSSERVAZIONE 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*.

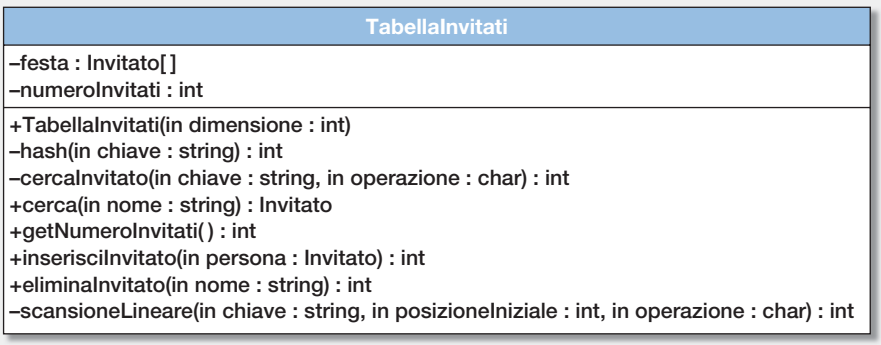


FIGURA **27**

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
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)
```

OSSERVAZIONE 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.

Sintesi

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.

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 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 effettuare 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; sia 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 il nodo 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 prescelto.

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 destro differisce al più di 1 dal numero di nodi del sottoalbero sinistro. 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).

QUESITI

1 Una lista è una struttura dati ad accesso …

A … diretto.
B … sequenziale.
C … sia diretto sia sequenziale.
D … inverso.

2 Il numero degli elementi di una lista è …

A … prefissato.
B … variabile.
C … maggiore di 1.
D … minore di un valore prefissato.

3 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 Nessuna delle risposte precedenti.

4 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 uniforme.

5 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.

6 In una lista multipla ogni nodo …

A … deve avere almeno due componenti con valenza di riferimento (*link*) distinti.
B … deve avere almeno due componenti con valenza informativa distinti.
C … deve avere più di due componenti con valenza di riferimento (*link*) distinti.
D Nessuna delle risposte precedenti.

7 Per la memorizzazione di un albero …

A … è sufficiente una lista semplice.
B … è necessaria una lista multipla.
C … è necessario un *array* multidimensionale.
D … è preferibile una tabella a indirizzamento *hash*.

8 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 … occorre una pila per i riferimenti ai nodi.
D … è possibile visitare un albero solo con un algoritmo ricorsivo.

9 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 Nessuna delle risposte precedenti.

10 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 Nessuna delle risposte precedenti.

11 La politica LIFO è propria di …

A … un *array* monodimensionale.
B … una tabella.
C … una pila.
D … una coda.

12 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.

13 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.

14 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.

15 La politica FIFO è propria di …

A … un *array* monodimensionale (vettore).
B … una pila.

C … una coda.

D … una lista.

16 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 Nessuna delle risposte precedenti.

17 Indicare lo scopo del seguente metodo Java di una 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;
}
```

18 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.

19 Che cosa restituisce il seguente metodo Java di una 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);
}
```

ESERCIZI

Liste

1 Scrivere un metodo Java denominato *shuffle* per una 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 $k \geq N$, con *N* numero di elementi della lista, deve essere posto in ultima posizione.

2 Scrivere un metodo Java denominato *lastFirst* per una ipotetica classe *Lista* che sposti in testa l'ultimo elemento e contemporaneamente sposti in coda il primo elemento.

3 Scrivere un costruttore per un'ipotetica classe Java *Lista* che inizializza 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.

4 Scrivere un metodo Java denominato *isSublist* per una ipotetica classe *Lista* che, a partire da un oggetto istanza di *Lista* fornito come argomento, verifichi se esso è o meno una sottolista.

5 Scrivere un costruttore per un'ipotetica classe Java *Lista* che inizializza 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.

6 Scrivere un metodo Java denominato *erase* per una 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.

7 In un dispositivo palmare abilitato alla ricezione della posta elettronica le mail sono mantenute in una lista ordinata in cui vengono inserite al momento della ricezione. Per ogni 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 una mail alla lista tenendo conto che l'ultima ricevuta è sempre la prima della lista;
- eliminazione di una mail data la sua posizione nella lista;
- ricerca di tutte le 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).

8 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 col 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

9 Scrivere un metodo Java per una 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.

10 Scrivere un metodo Java denominato *Boscaiolo* per una 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.

11 Scrivere un metodo Java ricerca per una ipotetica classe *AlberoBinario* che determini la presenza o meno nei nodi dell'albero di uno specifico valore fornito come argomento.

12 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*

13 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.

14 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.

15 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). 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.

LABORATORIO

1 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 (funzione *shuffle*: il metodo *nextInt* della classe *java.util.Random* restituisce un numero casuale intero).

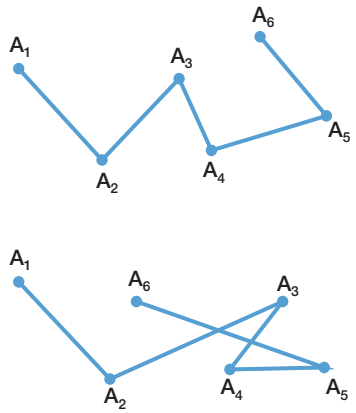
Realizzare un *main* che consenta all'utente di eseguire le operazioni in modo interattivo.

2  Un ufficio accetta le pratiche che deve svolgere da un sistema di immissione on-line accessibile dagli utenti mediante Internet: ogni pratica è caratterizzata da un numero progressivo assegnato automaticamente dal sistema, dal cognome e 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 utilizza 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 codi-

ce 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.

3 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, vengono 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.

4 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

thread
filamento, filatura

to depict
rappresentare tramite un disegno

head/tail
testa/coda

parenthesized
messo tra parentesi

sake
letteralmente: motivo, ragione

unreachable
non raggiungibile

to take care of
avere cura di

self-referential
autoreferenziale, che si autoriferisce

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.

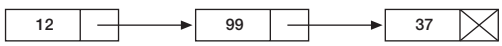
7.2 Cells and lists

7.2.1 Illustrating the concepts of cells and lists

Consider an arbitrary set of objects $O = \{O_1, \ldots, O_n\}$ 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 → 99 → 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 parenthesized 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 initializing 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 initialize 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).
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 tenere 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

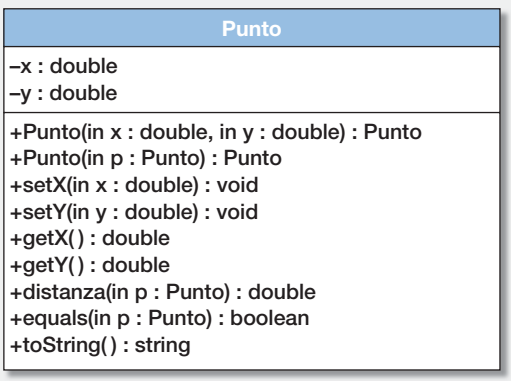


FIGURA 1

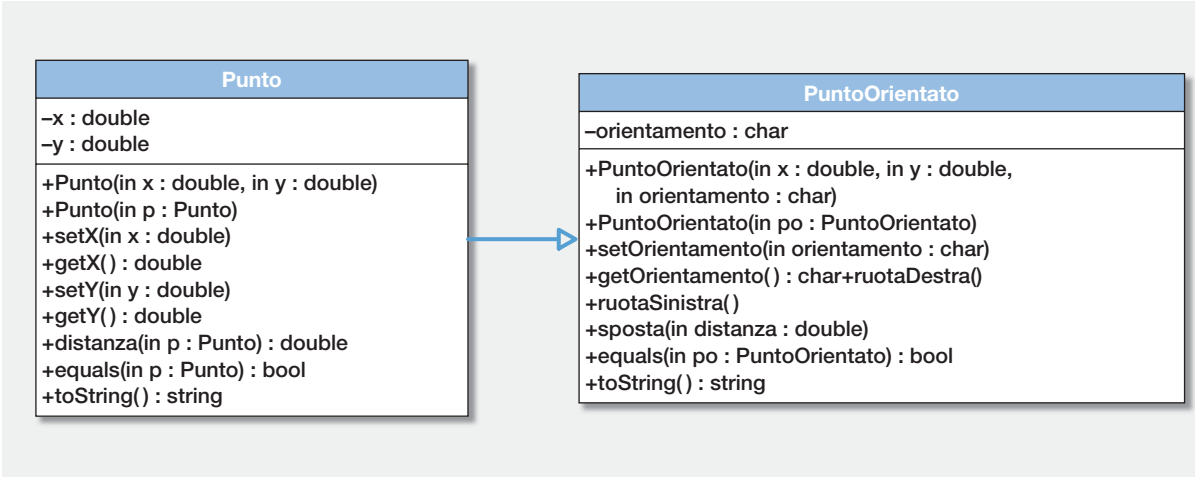


FIGURA **2**

«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.

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);
      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("P01 e P03 coincidono");
   else
      System.out.println("P01 e P03 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 P01-P03: 0.0
P01 e P03 coincidono
PO1 = (1.0;1.0)A
PO1 = (1.0;11.0)A
```

OSSERVAZIONE 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 eredità.

OSSERVAZIONE

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) e «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*.

OSSERVAZIONE Sia nella classe *Punto* sia nella classe derivata *PuntoOrientato* vi sono più costruttori: tutti hanno lo stesso nome e, come già abbiamo 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.

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.

La tecnica di ridefinizione in una classe derivata di metodi già presenti nella superclasse è denominata ***overridding***: 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.

OSSERVAZIONE 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 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*.

OSSERVAZIONE 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.

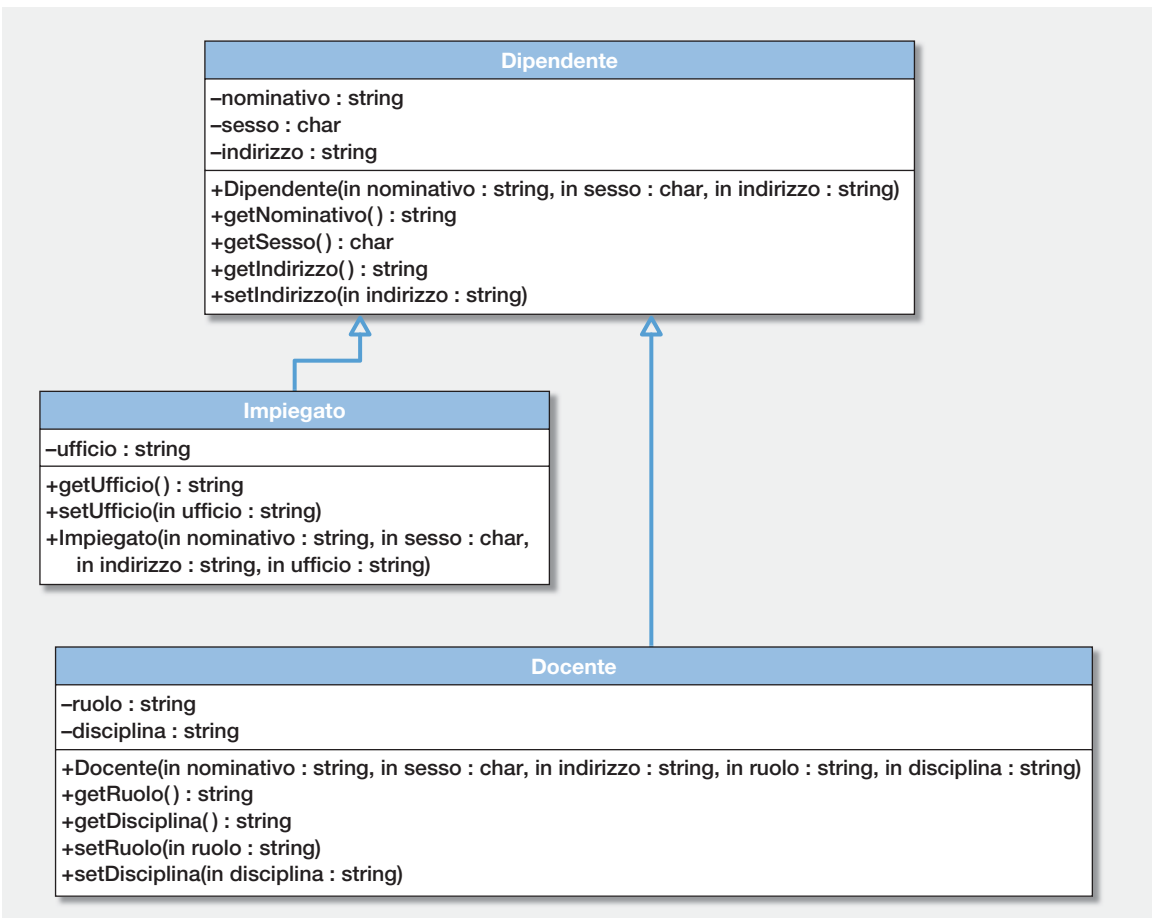


FIGURA **3**

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;
  }

  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;
  }
}
```

OSSERVAZIONE 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 ancora 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 *PuntoOrienta-*

to dell'oggetto *po* passato come argomento – utilizza i metodi pubblici *getX* e *getY* per accedervi.

OSSERVAZIONE

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.

OSSERVAZIONE 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'*overridding* 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.

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

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*).

OSSERVAZIONE 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 rispodere 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*.

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.

OSSERVAZIONE 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 (*runtime*) si può verificare un'eccezione di tipo *ClassCastException* se il 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(…);
```

OSSERVAZIONE 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'*overridding* degli attributi di una classe modificandone eventualmente anche il tipo nella classe derivata. In realtà in questo caso, più che di *overridding*, 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
```

OSSERVAZIONE 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. se i loro attributi corrispondenti assumono lo stesso valore se sono di tipo primitivo, o riferiscono lo stesso oggetto se sono riferimenti;
2. se i loro attributi corrispondenti assumono lo stesso valore se sono di tipo primitivo, o riferiscono oggetti che sono ricorsivamente uguali se sono riferimenti.

OSSERVAZIONE 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 tipologia 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 che 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.

OSSERVAZIONE La *deep copy* impone di copiare ogni eventuale oggetto riferito da un attributo e ogni singolo elemento degli *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 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) e, 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

Il seguente metodo *main* 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();
}
```

OSSERVAZIONE 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();
}
```

Si noti che l'eccezione catturata e non gestita nel codice del metodo *clone* non viene mai generata, in quanto il valore dell'attributo *orientamento* fornito come parametro al costruttore della classe *PuntoOrientato* assume sempre un valore corretto.

OSSERVAZIONE 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`, esso 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 A2, 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 {…}
```

OSSERVAZIONE 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 A2, 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;
}
```

OSSERVAZIONE Dopo aver eseguito una *deep copy* di un oggetto di classe *Mensola* invocandone il metodo *clone*, la situazione è quella schematizzata in 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**)!

OSSERVAZIONE 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.

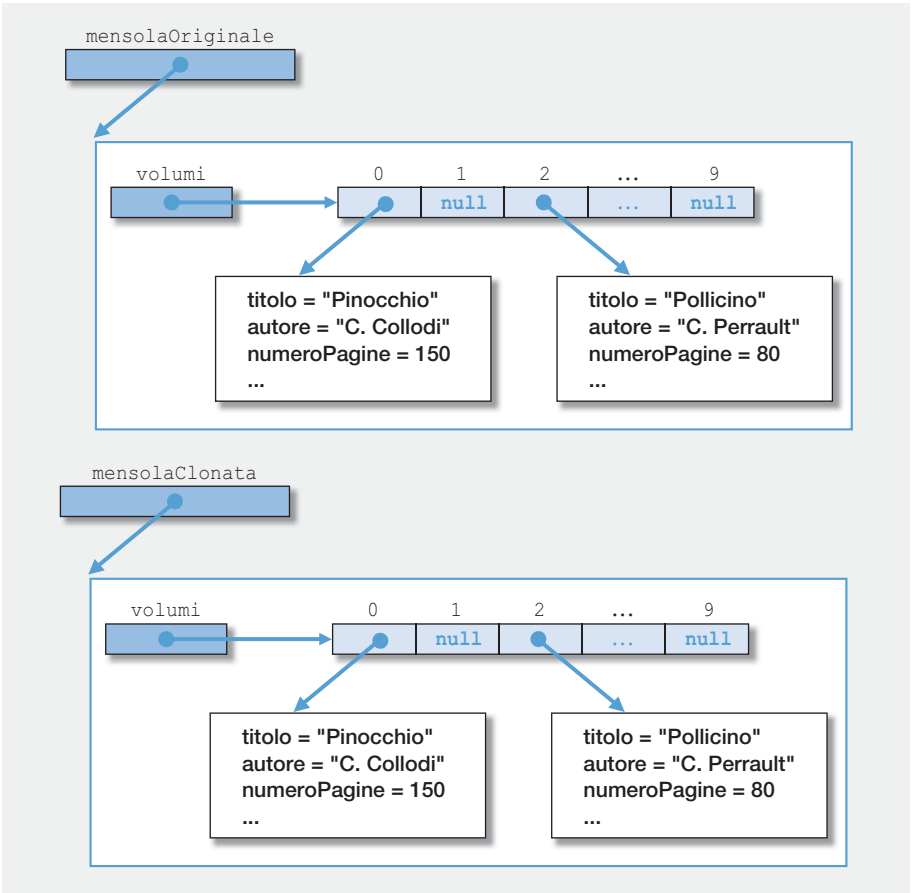


FIGURA 4

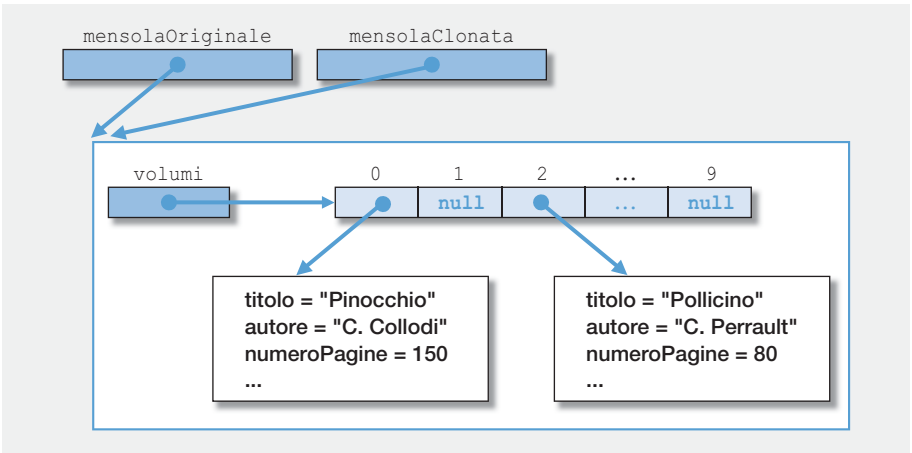


FIGURA 5

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");
…
```

OSSERVAZIONE 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 consentono di usare questo meccanismo di fattorizzazione degli attributi comuni anche nel caso in cui la superclasse risultante non sia «ben definita».
L'uso della 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 che 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 superficie() {
    return 4 * Math.PI * Math.pow(raggio,2);
  }

  public 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 peso() {
    return pesoSpecifico * volume();
  }
}
```

La classe astratta *Solido* fattorizza attributi e metodi comuni alle classi *Sfera* e *Cubo* (FIGURA **6**):

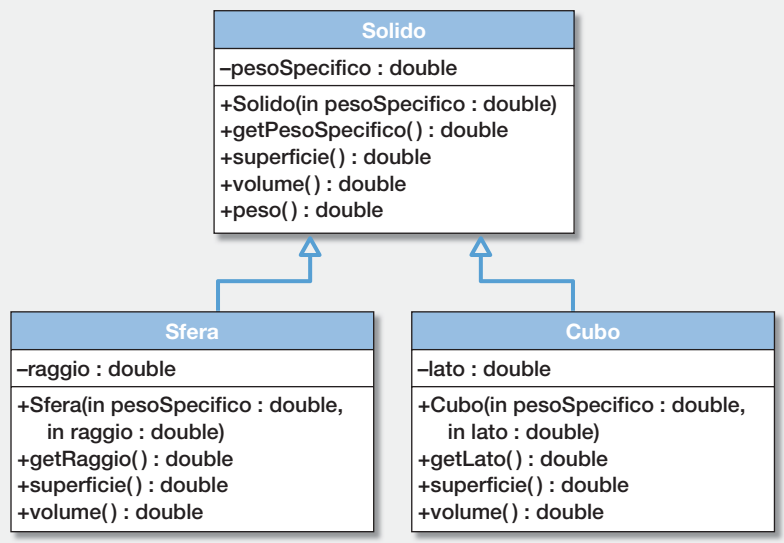


FIGURA **6**

```
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 {
  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);
  }
}
```

OSSERVAZIONE 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 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).

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.

Una interfaccia viene **implementata** dalle classi mediante l'uso della parola chiave `implements`.

OSSERVAZIONE 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>>`.

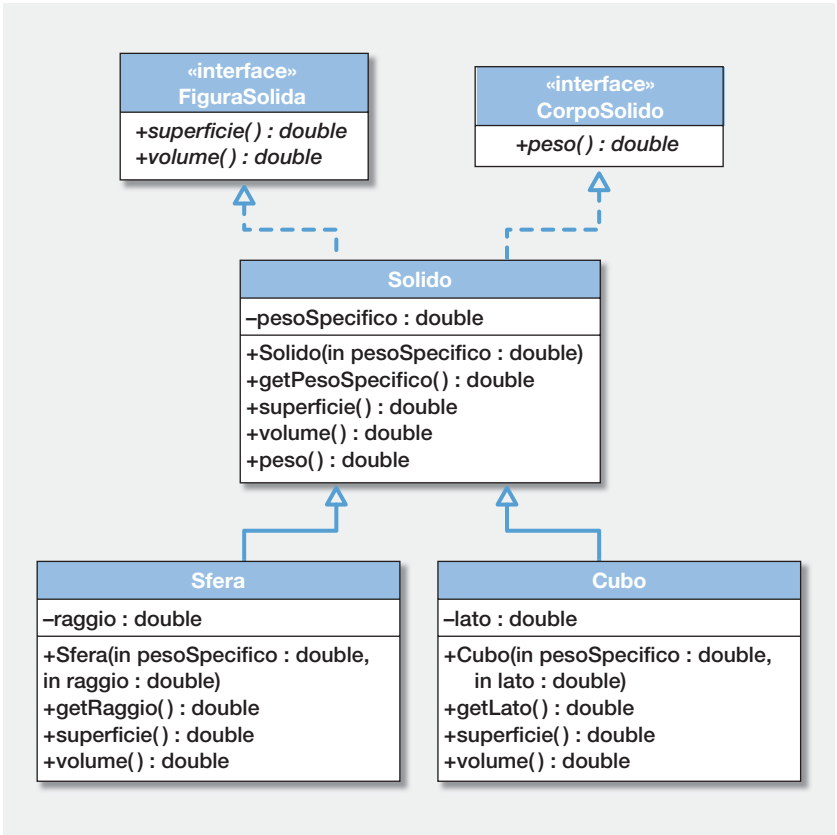


FIGURA **7**

Simulare l'eredità 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'eredità multipla.

OSSERVAZIONE

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'eredità multipla di classi.

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**).

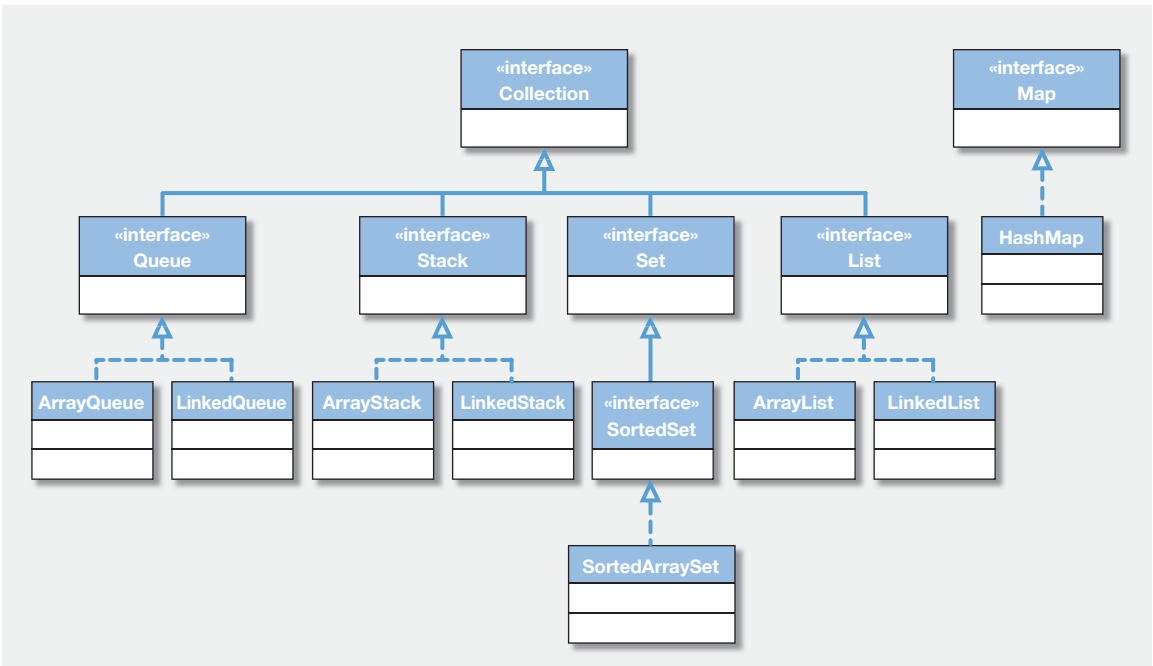


FIGURA **8**

Se un'interfaccia definisce attributi, essi sono implicitamente costanti e sono 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 Boltzman
  static final double R = 8.3143;       // costante dei gas perfetti
  static final double h = 6.62559E-34;  // costante di Plank
}
```

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.

OSSERVAZIONE 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.volume());
   Solido solido2 = cubo;
   System.out.println("Solido solido2-> Volume….: " + solido2.volume());
   System.out.println("Solido solido2-> Peso……: " + solido2.volume());
}
```

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 i metodi *volume* sono implementazioni 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.

OSSERVAZIONE 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.

OSSERVAZIONE 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 all'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.

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.

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.

OSSERVAZIONE

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.
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;
  }
```

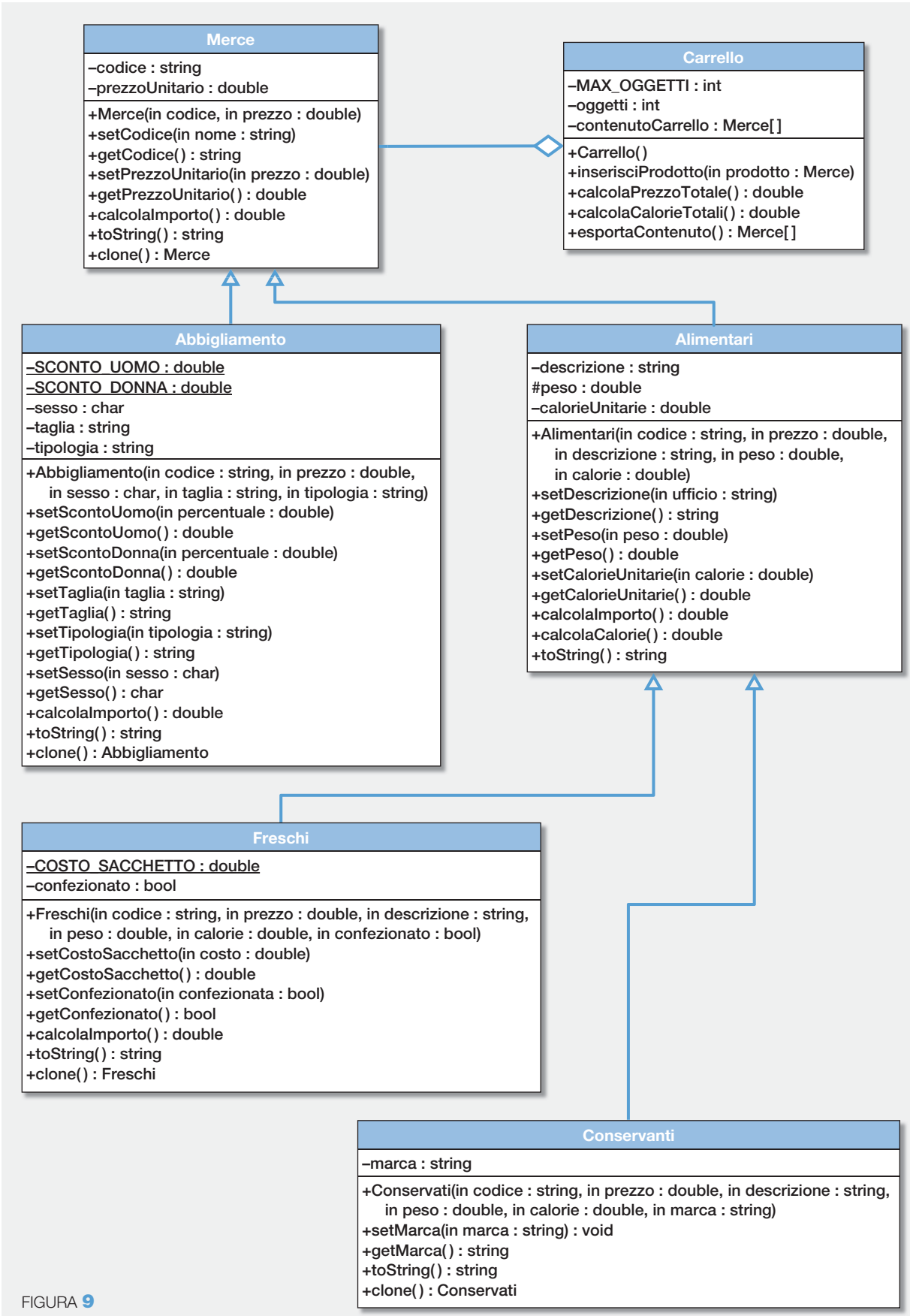

Merce
–codice : string
–prezzoUnitario : double
+Merce(in codice, in prezzo : double)
+setCodice(in nome : string)
+getCodice( ) : string
+setPrezzoUnitario(in prezzo : double)
+getPrezzoUnitario( ) : double
+calcolaImporto( ) : double
+toString( ) : string
+clone( ) : Merce
Carrello
–MAX_OGGETTI : int
–oggetti : int
–contenutoCarrello : Merce[ ]
+Carrello( )
+inserisciProdotto(in prodotto : Merce)
+calcolaPrezzoTotale( ) : double
+calcolaCalorieTotali( ) : double
+esportaContenuto( ) : Merce[ ]
Abbigliamento
–SCONTO_UOMO : double
–SCONTO_DONNA : 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
–COSTO_SACCHETTO : 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
Conservanti
–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**

```
  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 final double SCONTO_DONNA = 10;
  private final double SCONTO_UOMO = 5;
  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 SCONTO_DONNA;
  }

  static public void setScontoDonna(double percentuale) {
    SCONTO_DONNA = percentuale;
  }

  static public double getScontoUomo() {
    return SCONTO_UOMO;
  }

  static public void setScontoUomo(double percentuale) {
    SCONTO_UOMO = 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*SCONTO_DONNA/100.0;
    if (sesso == 'M')
      return prezzoUnitario-prezzoUnitario*SCONTO_UOMO/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;
  private 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();
  }
}
```

```
public class Freschi extends Alimentari {
  static private double COSTO_SACCHETTO = 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 COSTO_SACCHETTO;
  }

  static public void setCostoSacchetto(double costo) {
    COSTO_SACCHETTO = costo;
  }

  public void setConfezionato(boolean confezionato) {
    this.confezionato = confezionato;
  }

  public boolean getConfezionata() {
    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 += COSTO_SACCHETTO;
    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 alla 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();
  }
}
```

Come 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.formFile();
  }
  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();
  }
```

OSSERVAZIONE 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*.

OSSERVAZIONE Le eccezioni generate da parte delle sottoclassi possono solo essere un sottoinsieme delle eccezioni generate dalla superclasse.

Sintesi

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.

Overridding. 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 fare riferire un oggetto istanza di una certa classe da una variabile avente come tipo una qualsiasi superclasse, ma non il 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 (*runtime*) si può verificare un'eccezione di tipo *ClassCastException* se l'oggetto del *casting* non è compatibile col 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; in questo caso gli attributi del nuovo oggetto creato hanno lo stesso valore dell'oggetto originale, sia che si tratti di tipi primitivi che 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 (senza cioè 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). Una 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'eredità 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.

QUESITI

1 **L'ereditarietà è un …**

A … meccanismo di astrazione finalizzato alla definizione di interfacce.

B … meccanismo di astrazione finalizzato all'implementazione dell'*information hiding*.

C … meccanismo di astrazione finalizzato alla costruzione di gerarchie classi.

D … meccanismo di astrazione finalizzato alla gestione degli attributi complessi.

2 **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.

3 **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.

4 **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.

5 **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*.

6 **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 Nessuna delle risposte precedenti.

7 **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*.

8 **Il metodo *equals* …**

A … serve a confrontare due oggetti per verificare se i loro attributi hanno lo stesso valore.

B … serve a confrontare due oggetti per verificare se sono o meno lo stesso oggetto.

C … serve a confrontare due classi per verificare se sono identiche.

D … serve a confrontare gli attributi corrispondenti di due oggetti distinti.

9 **Il metodo *toString* …**

A … è un metodo della classe *Object* che serve a convertire una classe in una stringa.

B … è un metodo della classe *Object* che serve a ottenere una stringa che descrive l'elenco delle classi di una gerarchia.

C … è un metodo della classe *Object* che serve a ottenere l'elenco degli identificatori degli attributi di una classe.

D … è un metodo della classe *Object* che serve a ottenere l'elenco degli identificatori dei metodi di una classe.

10 **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 Nessuna delle risposte precedenti.

11 **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.

12 **È possibile fare 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*.

13 **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*.

14 **Il *casting* implicito si ha quando questo avviene …**

A … verso una superclasse della classe dell'oggetto riferito.

B … verso una sottoclasse della classe dell'oggetto riferito.

C … verso la classe dell'oggetto riferito.

D … verso una qualsiasi classe se non si specifica l'operatore di *casting*.

15 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 Nessuna delle risposte precedenti.

16 Il *down-casting* permette di trasformare …

A … un riferimento dalla classe radice della gerarchia verso una sottoclasse.
B … un riferimento di una sottoclasse verso la classe radice della gerarchia.
C … un riferimento da una qualsiasi classe a una qualsiasi altra classe.
D … un riferimento qualsiasi verso la classe *Object*.

17 Che cosa 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*.

18 Indicare quali delle seguenti affermazioni sono vere relativamente a una classe astratta in linguaggio Java.

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.

19 Indicare quali delle seguenti affermazioni sono vere relativamente a un'interfaccia in linguaggio Java.

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.

20 Indicare quali delle seguenti affermazioni sono vere relativamente al polimorfismo in linguaggio Java.

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 da dichiarazione degli attributi.
D Nel linguaggio Java il polimorfismo è basato sul *binding* statico.

21 Il polimorfismo si applica a …

A … classi che non sono in relazione gerarchica.
B … classi che sono in relazione gerarchica.
C … esclusivamente classi astratte.
D Nessuna delle risposte precedenti.

22 L'acronimo RTTI significa …

A *Run-Time Type Identification*.
B *Run-Time Trouble Identification*.
C *Run-Time Type Implementation*.
D *Run-Type Time Identification*.

23 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*.

24 **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.

ESERCIZI

1 **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
                        e': " + b.q);
  }
}
```

2 **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 infine 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.**

3 **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.

4 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.

5 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 col 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.

6 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 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.

7 Si intende 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 avere 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.

8 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 che 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 avere 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.

9 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 col pieno di carburante e che il costo del noleggio è così calcolato:

- autovetture: 50 euro al giorno, più 1 euro ogni 25 km percorsi, più 2 euro per ogni litro di carburante che manca dal pieno al momento della restituzione;
- furgoni: 70 euro al giorno, più 1 euro ogni 30 km percorsi dopo i primi 100 km, più 2 euro 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.

10 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 più 6 volte l'esperienza
Elfi	Se l'esperienza è inferiore a 5 la forza è 20 più 3 volte l'esperienza, altrimenti è 80 + 2 volte l'esperienza
Nani	20 più 4 volte l'esperienza
Hobbit	10 più 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 più 5 volte l'esperienza
Sudroni	40 più 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 e 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 modellano lo scenario descritto;
- il codice Java che implementa 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.

11 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.

LABORATORIO

1 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 avere 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.

2 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 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 nel menu. Si richiede:

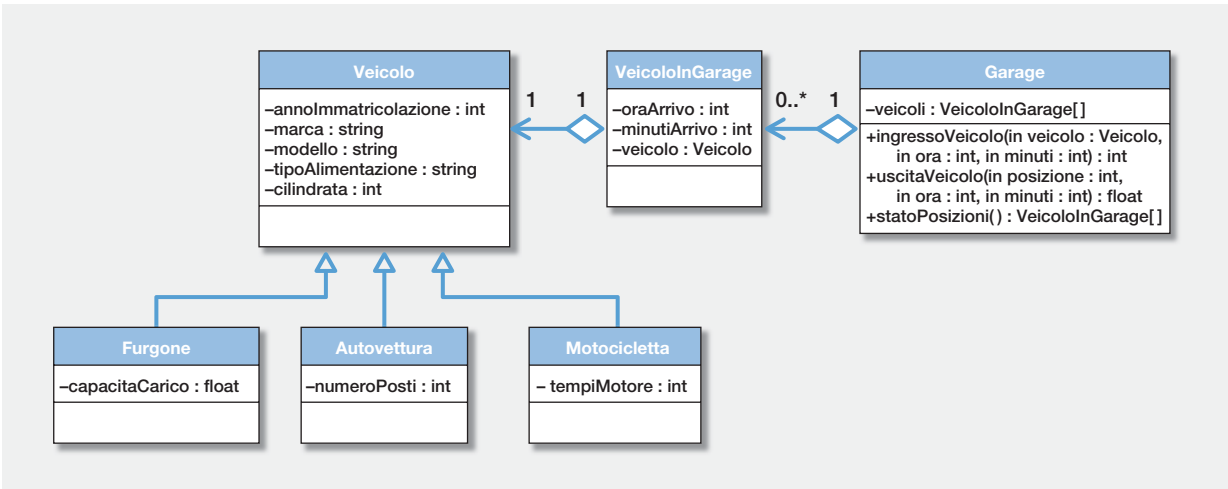


FIGURA **11**

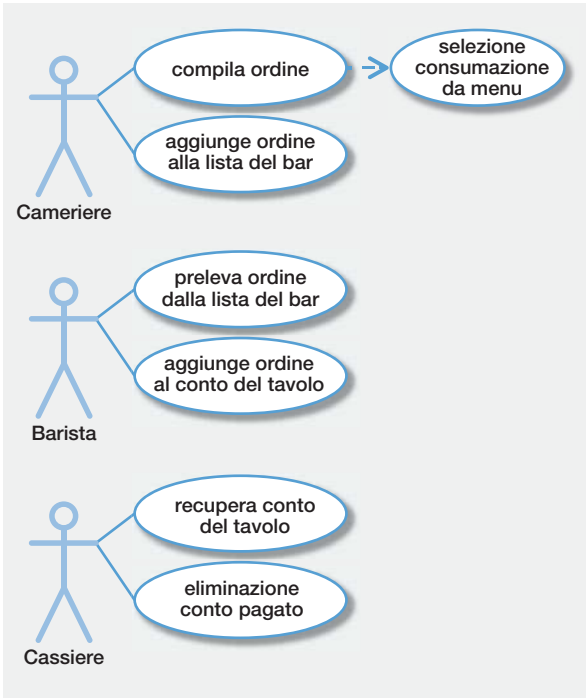


FIGURA **12**

- 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**).

3 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.

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
```


inheritance
ereditarietà

trivial
triviale, banale

leap-year
anno bisestile

goal
obiettivo

built-in
incorporato

drawback
inconveniente

relationship
relazione

to underline
sottolineare

```
{
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 And Bartlett Publishers, 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?

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.

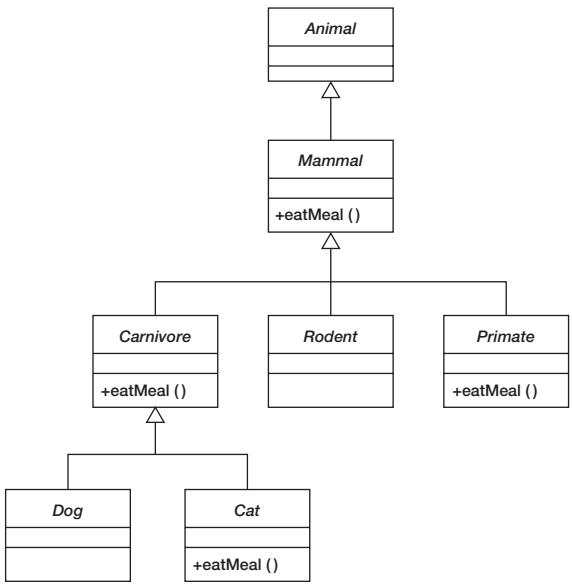


Figure 2.5. Polymorphism means a Cat uses its own eatMeal, a Dog uses the Carnivore eatMeal, and a Rodent the Mammal eatMeal.

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, 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**.

to behave
comportarsi

mammal
mammifero

rodent
roditore

primate
primati
(uomini, scimmie)

habit
abitudine

meal
pasto

ORIGINAL DOCUMENT

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.

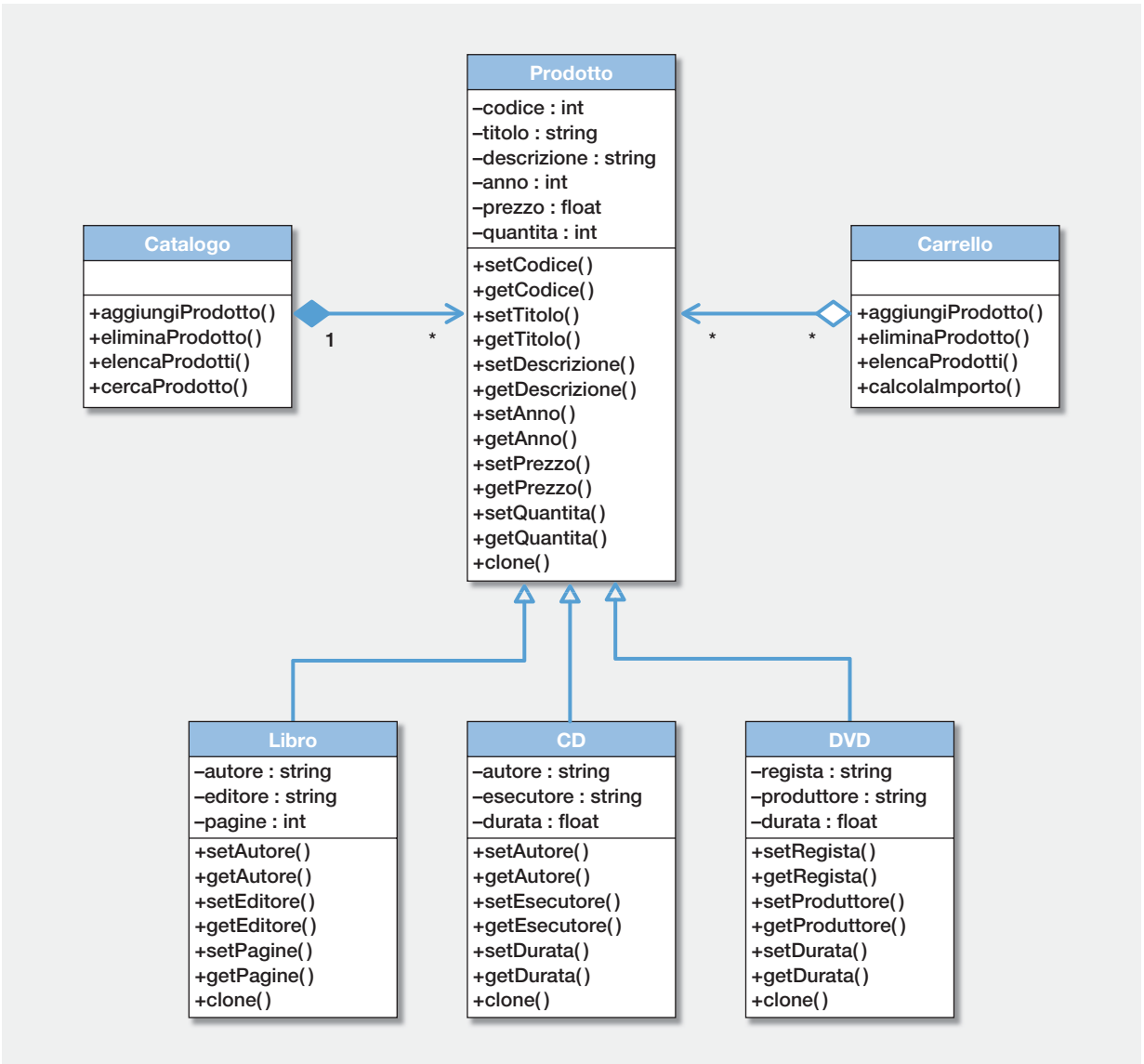


FIGURA **1**

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;
  }
}
```

OSSERVAZIONE 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.

OSSERVAZIONE 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.

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 una 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;
```

▶

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 *generic* Java 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 precedentemente alla versione 5 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 Coppia(T1 primo, T2 secondo) {
    this.primo = primo;
    this.secondo = secondo;
  }

  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*.

OSSERVAZIONE 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*:

```
// 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<T>(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();
  }
}
```

OSSERVAZIONE 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<Persona>();
```

```
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<Punto>();
coda.enqueue(p1);
coda.enqueue(p2);
…
Punto p = coda.dequeue();
…
```

OSSERVAZIONE Prima della versione 5 il linguaggio di programmazione 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: questa 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 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, che ha introdotto le 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.
Alcuni errori di programmazione molto comuni comportano un uso non desiderato (e non necessario) dei tipi *raw*; in particolare è frequente dimenticarsi di istanziare un tipo parametrico nell'invocazione del costruttore di una classe generica

```
Coda<Punto> c = new Coda();
```

In questo caso il compilatore segnala l'errore con un semplice «*unchecked warning*» senza impedire la creazione del codice eseguibile.

ESEMPIO

A partire dalla precedente definizione della classe generica *Coda<T>* il seguente codice causa un errore di compilazione per l'incoerenza del tipo *Punto* rispetto al tipo *Persona*:

```
…
Persona p1 = new Persona("Bianchi Giovanni", 'M');
Punto p2 = new Punto(1., 0.);

Coda<Persona> coda = new Coda<Persona>();
coda.enqueue(p1);
coda.enqueue(p2); // errore rilevato in fase di compilazione
…
```

mentre il codice che segue viene compilato correttamente e l'errore causa la generazione di un'eccezione in esecuzione:

```
…
Persona p1 = new Persona("Bianchi Giovanni", 'M');
Punto p2 = new Punto(1., 0.);

Coda coda = new Coda(); // classe generica usata come tipo raw
coda.enqueue(p1);
coda.enqueue(p2); // eccezione generata in fase di esecuzione
…
```

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 la seguente sintassi:

Tipo parametrico	Vincolo
`<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* è super-tipo della classe *C*

ESEMPIO

Una classe generica *Coppia<T1,T2>* per la quale si intende vincolare gli elementi della coppia a essere di tipo numerico (*Integer*, *Double*, …) deve essere definita nel seguente modo:

```
public class CoppiaNumerica<T1 extends Number, T2 extends Number> {
  private T1 primo;
  private T2 secondo;

  public CoppiaNumerica(T1 primo, T2 secondo) {
    this.primo = primo;
    this.secondo = secondo;
  }
  …
  …
  …
}
```

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);
  }
}
```

OSSERVAZIONE Come 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 a questo punto complesso 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<Invitato> 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 della 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 tipicamente 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 tipicamente 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

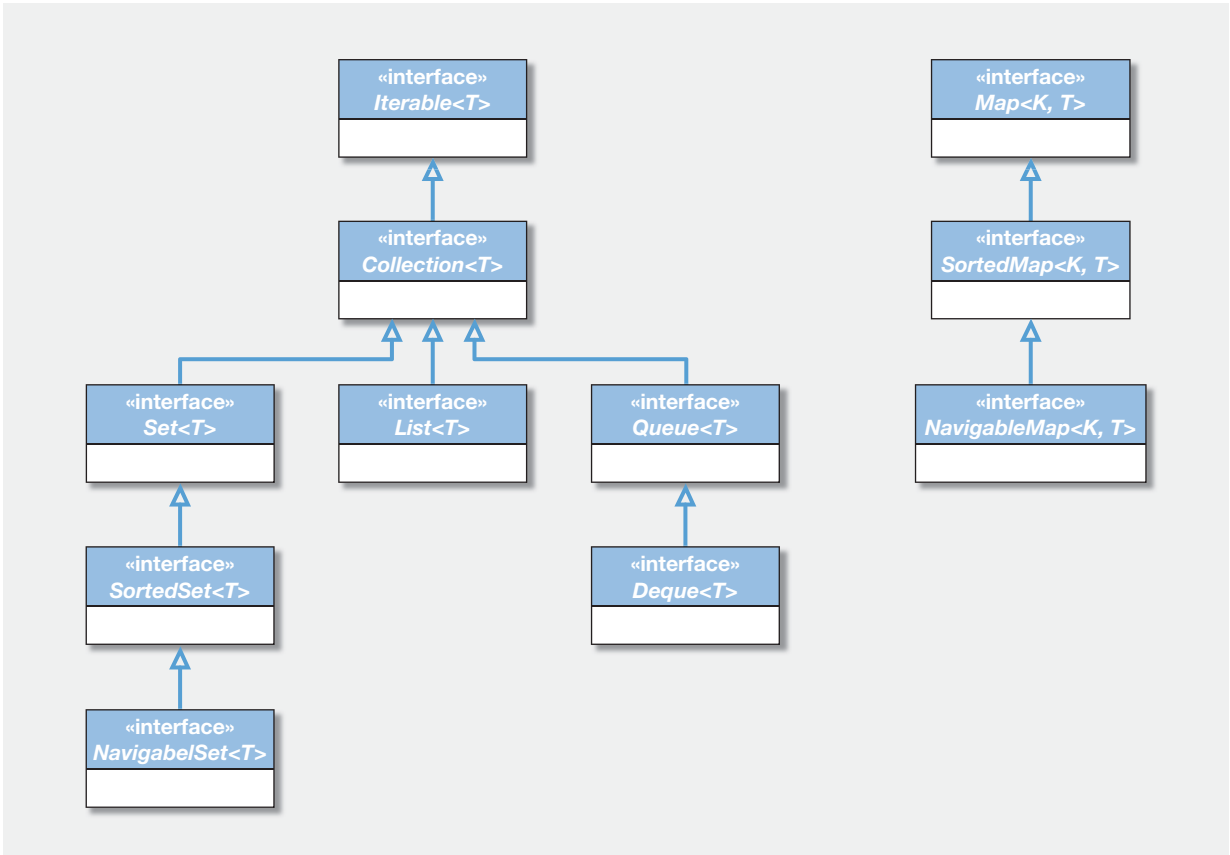


FIGURA **2**

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).

OSSERVAZIONE 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. Come 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<Punto>();
  }
  …
  …
  …
  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) {
  Iteratr<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**).
Le classi generiche concrete presenti nel diagramma UML della 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.

OSSERVAZIONE I nomi delle classi evidenziano la loro implementazione interna: *TreeSet<T>* e *TreeMap<T>* sono alberi bilanciati, *HashSet<T>* e *HashMap<T>* sono tabelle hash, *ArrayList<T>* è un vettore dinamico e *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.

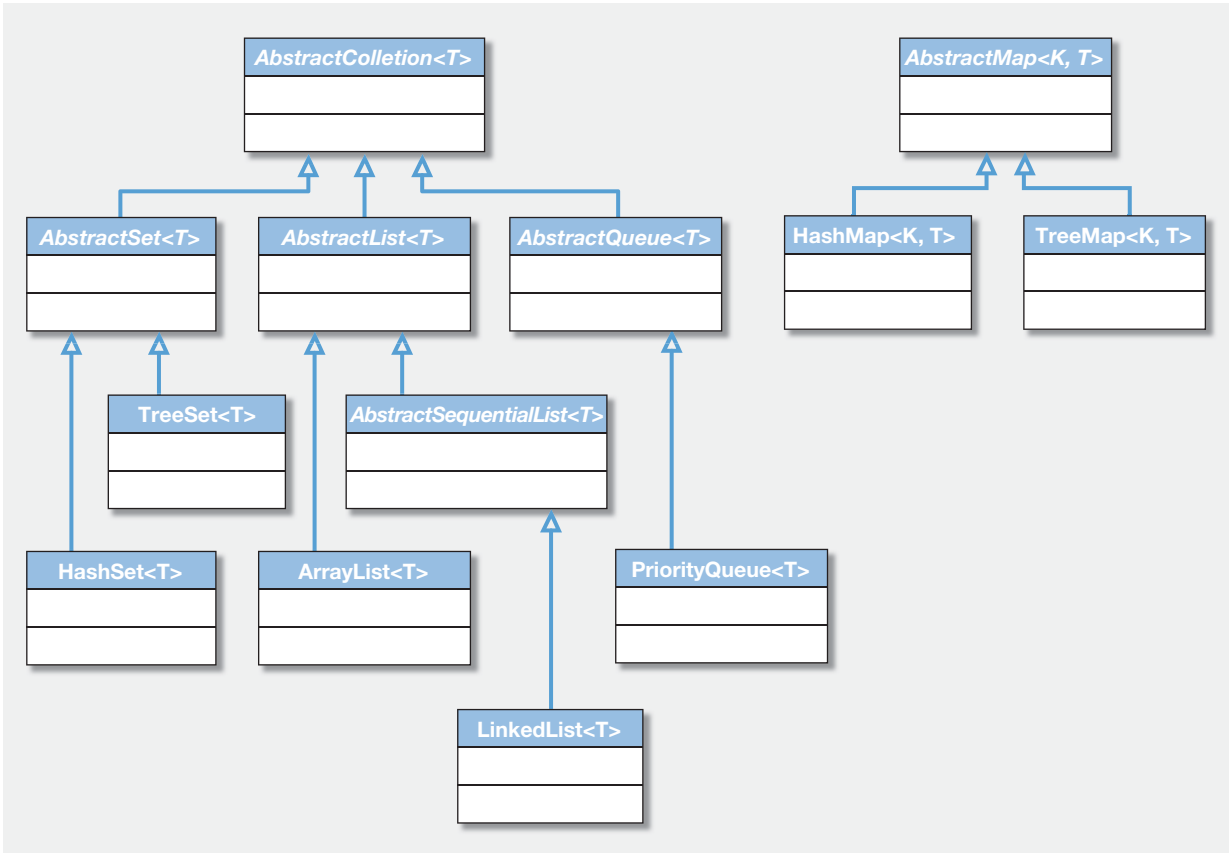


FIGURA **3**

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 ▶

▶ TABELLA 2

Firma del metodo	Descrizione della funzionalità
`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].

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* ▶

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.

▶ TABELLA 3

Firma del metodo	Descrizione della funzionalità
`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*

OSSERVAZIONE 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'impossibilità di avere contenitori generici di tipi primitivi, ma la presenza delle classi *wrapper* non rende particolarmente grave questo problema.

4. M. Naftalin, P. Walder, *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*

OSSERVAZIONE 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:

- 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<Dipendente>(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
  }
```

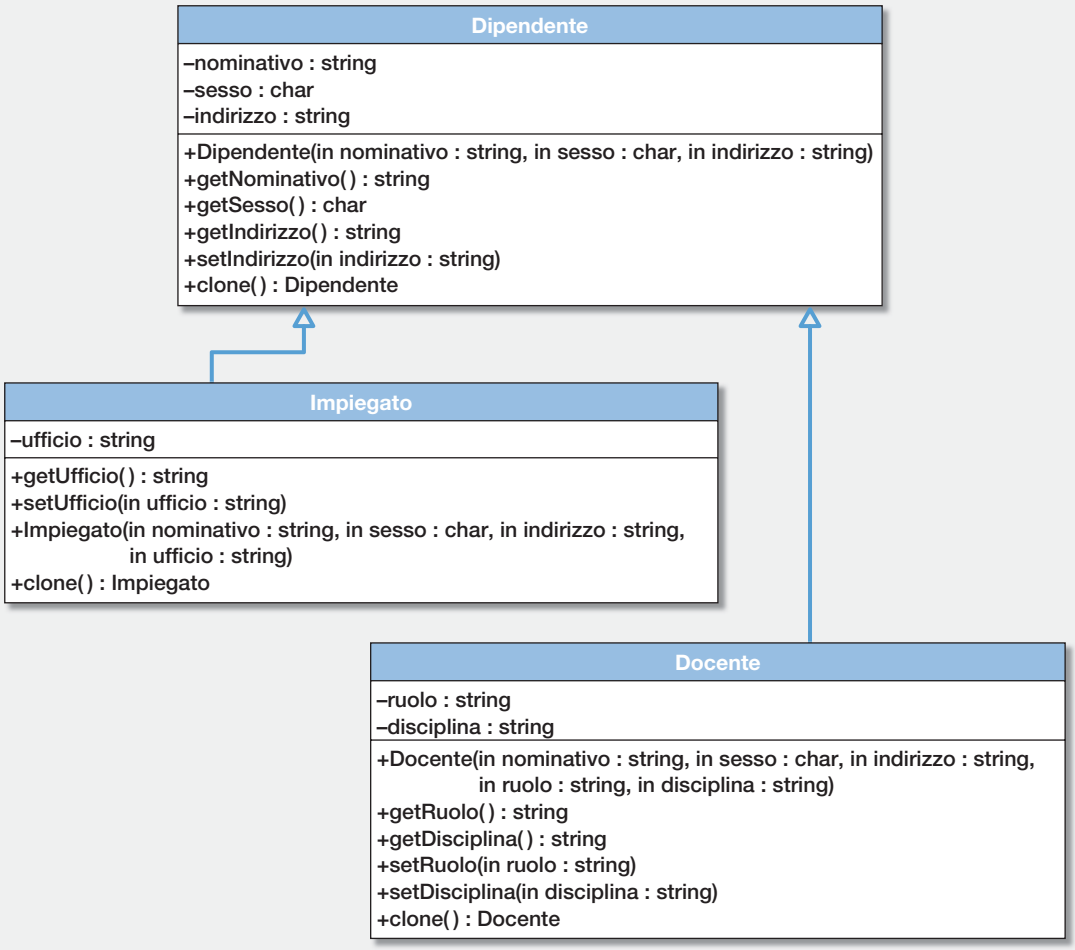


FIGURA **4**

```
// 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;
  }
}
```

OSSERVAZIONE 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: tipicamente 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 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 di cui 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, P. Walder, *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)

OSSERVAZIONE 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.

La seguente classe Java implementa le funzionalità richieste memorizzando i libri in un contenitore di tipo *TreeMap<T>*: ▶

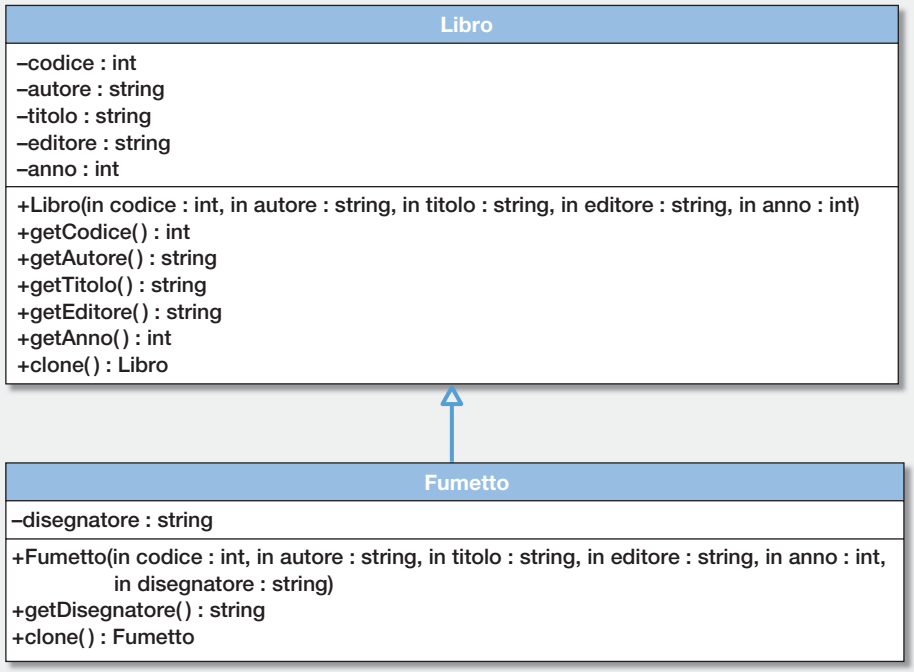


FIGURA **5**

```
class LibroEsistente extends Exception {
}
class LibroNonEsistente extends Exception {
}

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;
  }
}
```

OSSERVAZIONE 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 *HahsMap<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 sono generici: 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<Persona>(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);
}
```

Sintesi

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 intende riferire intere famiglie di tipi piuttosto che un singolo tipo. Le clausole prevista 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 di cui 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, …

QUESITI

1 Un tipo parametrico del linguaggio Java è …

A … l'identificatore di tipo di un qualsiasi parametro passato a un qualsiasi metodo di una classe.

B … l'identificatore di tipo che nella definizione di una classe non viene esplicitato, ma che deve essere istanziato al momento della dichiarazione di un riferimento alla classe.

C … l'identificatore di tipo da specificare all'atto della definizione di una gerarchia di classi che permette di parametrizzare il tipo di ereditarietà.

D … l'identificatore del tipo alternativo da utilizzare per la compilazione di una classe in caso di errori causati dal tipo predefinito.

2 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.

3 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 Nessuna delle risposte precedenti.

4 Un tipo *raw* corrisponde a …

A … una classe generica utilizzata senza istanziarne i tipi parametrici.

B … una classe parametrica utilizzata per istanziare un tipo generico.

C … un qualsiasi tipo parametrico.

D … il tipo *Object*.

5 Indicare quali delle seguenti affermazioni sono vere relativamente alle classi generiche.

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.

6 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.

7 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.

8 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 Nessuna delle risposte precedenti.

9 **L'interfaccia Java *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 Nessuna delle risposte precedenti.

10 **Per l'impiego 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>*.

11 **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>*.

12 **Indicare quali delle seguenti affermazioni sono vere per la classe *Collections*.**

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.

ESERCIZI

1 **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.**

2 **Per contrastare i furti di motoveicoli e il conseguente uso di targhe falsificate si intende realizzare un archivio nazionale dei motoveicoli che possa essere consultato per verificare l'autenticità delle targhe. Dopo avere progettato mediante un diagramma UML una classe che modelli l'entità «motoveicolo», implementare in Java una classe che, utilizzando un contenitore del *Collection framework* del linguaggio, consenta la gestione completa dell'archivio e in particolare le seguenti operazioni:**

- **aggiunta di un nuovo motoveicolo;**
- **variazione dei dati relativi a un motoveicolo già inserito;**
- **eliminazione di un motoveicolo precedentemente inserito;**
- **visualizzazione dei dati relativi a un motoveicolo di cui è nota la targa;**
- **salvataggio su file dell'archivio dei motoveicoli;**
- **caricamento da file dell'archivio dei motoveicoli.**

3 **Si deve realizzare una rubrica per un dispositivo palmare per la gestione dei numeri telefonici dei contatti. Dopo avere 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 di e-mail;**

- eliminazione di un contatto specificando l'indirizzo di e-mail.

4 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.

5 Un corriere internazionale che serve esclusivamente clienti abbonati è organizzato in modo che, dal 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 avere 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.

6 Il call-center di una grande società di servizi è organizzato in modo 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 avere 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.

LABORATORIO

1 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 automati-

camente assegnato un codice di operazione) e ne realizzano la gestione realizzando 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.

2 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 realizzando 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.

3 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 un nuova autovettura nell'area specificandone la postazione,
  b) uscita dall'area di una 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 avere 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 un 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.

4 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.

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;
```

synergistic
sinergico

motto
motto, massima morale o proverbiale

to belong
appartenere

backed
sostenuto
appoggiato

by the way
a proposito

slightly
leggermente

arguably
molto probabilmente

perhaps
forse

trivial
banale

misnomer
improprio
designazione erronea

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, 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

Introduzione alle *Graphic User Interface* in Java

Si intende realizzare un programma Java da utilizzare come calcolatrice: il punto di inizio è una classe che consenta di applicare le quattro operazioni aritmetiche (somma, sottrazione, moltiplicazione e divisione) ciascuna a due operandi; sono necessari i seguenti attributi:

- *operando*: rappresenta l'ultimo operando inserito (quello normalmente visibile nel display della calcolatrice);
- *memoria:* è la variabile per il secondo operando da usarsi come «accumulatore» dei risultati;
- *operazione:* simbolo dell'operazione aritmetica di cui è richiesta l'esecuzione («+», «–», «*», «/»);
- *errore:* variabile booleana che registra l'eventuale stato di errore verificato nel corso di un calcolo (per esempio: una divisione per zero);

e, oltre al costruttore, i seguenti metodi:

- *get/set* per gli attributi *operazione*, *operando* e *memoria*;
- *cambiaSegno* per invertire il segno all'operando corrente;
- *cancella* per eseguire il reset della calcolatrice azzerando memoria e operando corrente e l'eventuale condizione di errore;
- *cancellaUltimoOperando* per cancellare solo l'ultimo operando inserito, cioè quello che è normalmente visualizzato nel display, e recuperare una eventale situazione di errore;
- *addiziona*, *sottrai*, *moltiplica* e *dividi* per richiedere le quattro operazioni fondamentali dell'aritmetica;
- *uguale* per eseguire l'ultima operazione richiesta e calcolarne il risultato;
- *risultato* per restituire in forma di stringa il valore normalmente visualizzato nel display: se l'ultima operazione impostata/eseguita ha generato un errore viene restituita la stringa «ERR».

ESEMPIO

Nel linguaggio di programmazione Java la classe *Calcolatrice* può essere così implementata:

```
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
  private void addiziona() {
    if (!errore) {
      setOperando(getMemoria()+operando);
      setMemoria(getOperando());
    }
  }

  //sottrazione
  private void sottrai() {
    if (!errore) {
      setOperando(getMemoria()-operando);
      setMemoria(getOperando());
    }
  }

  // moltiplicazione
  private void moltiplica() {
    if (!errore) {
      setOperando(getMemoria()*operando);
      setMemoria(getOperando());
    }
  }

  // divisione
  private 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);
  }
}
```

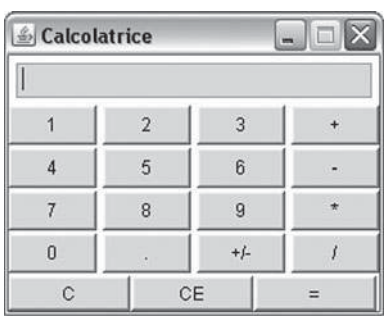
Il seguente metodo *main* consente di verificare la corretta funzionalità dei metodi della classe *Calcolatrice*:

```
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();
}
```

Producendo il seguente output:

```
30.0
ERR
10.0
1.0
```

Anche dotando la classe *Calcolatrice* dell'esempio precedente di un metodo *main* che mediante la visualizzazione di un menu richieda all'utente l'inserimento di operandi e operatori, l'uso di uno strumento software di questo tipo non è molto pratico! Ciò che si attende l'utente di un computer è uno strumento che disponga di un'interfaccia grafica che ne consenta un uso più immediato:



L'obiettivo di questo capitolo è presentare i principi fondamentali per la realizzazione di **interfacce grafiche utente** (**GUI, *Graphical User Interface***) utilizzando il linguaggio di programmazione Java; l'argomento è approfondito nel capitolo disponibile on-line e dedicato all'ambiente integrato di sviluppo *Netbeans*.

1 La libreria AWT: componenti fondamentali e gestione degli eventi

Una delle principali problematiche che i progettisti del linguaggio Java hanno dovuto affrontare è stata quella relativa alla possibilità di realizzare GUI in grado di funzionare correttamente e in modo omogeneo anche su piattaforme hardware e software molto diverse tra loro. Nel 1996 fu introdotto un package grafico denominato **AWT** (***Abstract Window Toolkit***) che mette in corrispondenza i componenti grafici specifici del sistema operativo ospite con apposite classi denominate *peer* e codificate prevalentemente nel linguaggio nativo della piattaforma di esecuzione.

Quando un programmatore utilizza un componente grafico AWT nella GUI di un'applicazione Java viene posizionato sullo schermo un oggetto grafico della piattaforma ospite: le istanze delle classi della libreria AWT provvedono a inviare le invocazioni dei metodi effettuate dall'applicazione sull'oggetto corrispondente della GUI. Allo stesso tempo, quando l'utente dell'applicazione interagisce con un elemento dell'interfaccia grafica, viene creato uno specifico oggetto *Event*, che viene inoltrato al corrispondente componente AWT, in modo tale che il programmatore possa gestirlo in una modalità *object oriented* e in modo completamente indipendente dalla piattaforma di esecuzione.

OSSERVAZIONE Le scelte progettuali in base alle quali è stata sviluppata la libreria AWT hanno comportato la disponibilità di un insieme estremamente limitato di componenti grafici comuni a tutti i sistemi operativi per i quali esisteva una JVM. Questo, se da un parte garantisce la compatibilità delle interfacce grafiche delle applicazioni eseguite su diverse piattaforme, costituisce un limite, in termini di potenza e flessibilità, rispetto alle aspettative degli sviluppatori software. Inoltre questa soluzione presenta un grave inconveniente: le applicazioni Java con GUI sviluppata utilizzando i componenti grafici AWT hanno un aspetto, e talvolta un comportamento, diverso in funzione della JVM su cui sono eseguite: spesso le interfacce grafiche realizzate per una particolare piattaforma mostrano gravi difetti se eseguite su una piattaforma differente, e in alcuni casi estremi risultano inutilizzabili. Questa problematica, ormai da tempo superata con il rilascio del *framework Swing* – è una delle principali ragioni per cui il motto dei creatori di Java –«scrivilo una volta sola ed eseguilo ovunque» è stato per molti anni noto come «scrivilo una volta sola e correggilo ovunque»!

AWT, *Swing* e SWT

La libreria AWT costituisce il primo approccio dei progettisti del linguaggio Java alla realizzazione di interfacce grafiche.
Nel 1999, con l'uscita della versione 1.2 del JDK, è stata sostituita dal *framework Swing*, i cui componenti grafici sono codificati completamente in linguaggio Java. Gli oggetti grafici *Swing* derivano dai corrispettivi componenti AWT: è quindi possibile utilizzare oggetti *Swing* dove erano previsti i corrispondenti oggetti AWT.
Swing impiega alcuni elementi AWT per il disegno di figure e la gestione degli eventi. Il codice di rappresentazione grafica dei componenti *Swing* è lo stesso per tutte le JVM dei vari sistemi operativi grafici: un pulsante *Swing* sullo schermo di un PC Windows avrà esattamente lo stesso aspetto sullo schermo di un Apple Mac, o utilizzando il sistema operativo Linux.
Questa soluzione risolve i problemi di uniformità visuale presenti in AWT e in altri *framework* grafici. Nel 2001 è stato rilasciato il *framework* grafico SWT (*Standard Widget Toolkit*) che integra le caratteristiche di *Swing* e di AWT adattando automaticamente l'aspetto grafico di un'applicazione Java a quello della piattaforma di esecuzione.

La libreria grafica AWT si fonda su due elementi principali:

- **Componenti**: oggetti aventi una rappresentazione grafica. La caratteristica principale di un componente è quella di essere un elemento con il quale l'utente può interagire; esempi di componenti sono i pulsanti (la pressione di un pulsante utilizzando il mouse può avviare un'azione), caselle di testo (la pressione dei tasti dei caratteri in corrispondenza di una casella di testo provoca la visualizzazione dei simboli corrispondenti), …
- **Contenitori**: oggetti che possono contenere altri elementi. La funzione principale di un contenitore è quella di permettere il posizionamento e il dimensionamento di componenti al proprio interno: queste operazioni possono essere eseguite in varie modalità, in funzione del gestore di layout (*layout manager*) associato al contenitore stesso. Esempi di contenitori sono le finestre (una finestra è un'area rettangolare con la barra del titolo e i pulsanti di ridimensionamento e chiusura), i pannelli (un pannello è un contenitore che può comprendere al proprio interno vari componenti posizionati, per esempio, all'interno di una finestra, o di un diverso pannello), …

Nella bozza di diagramma delle classi UML di FIGURA **1** è schematizzata la gerarchia delle principali classi della libreria AWT.

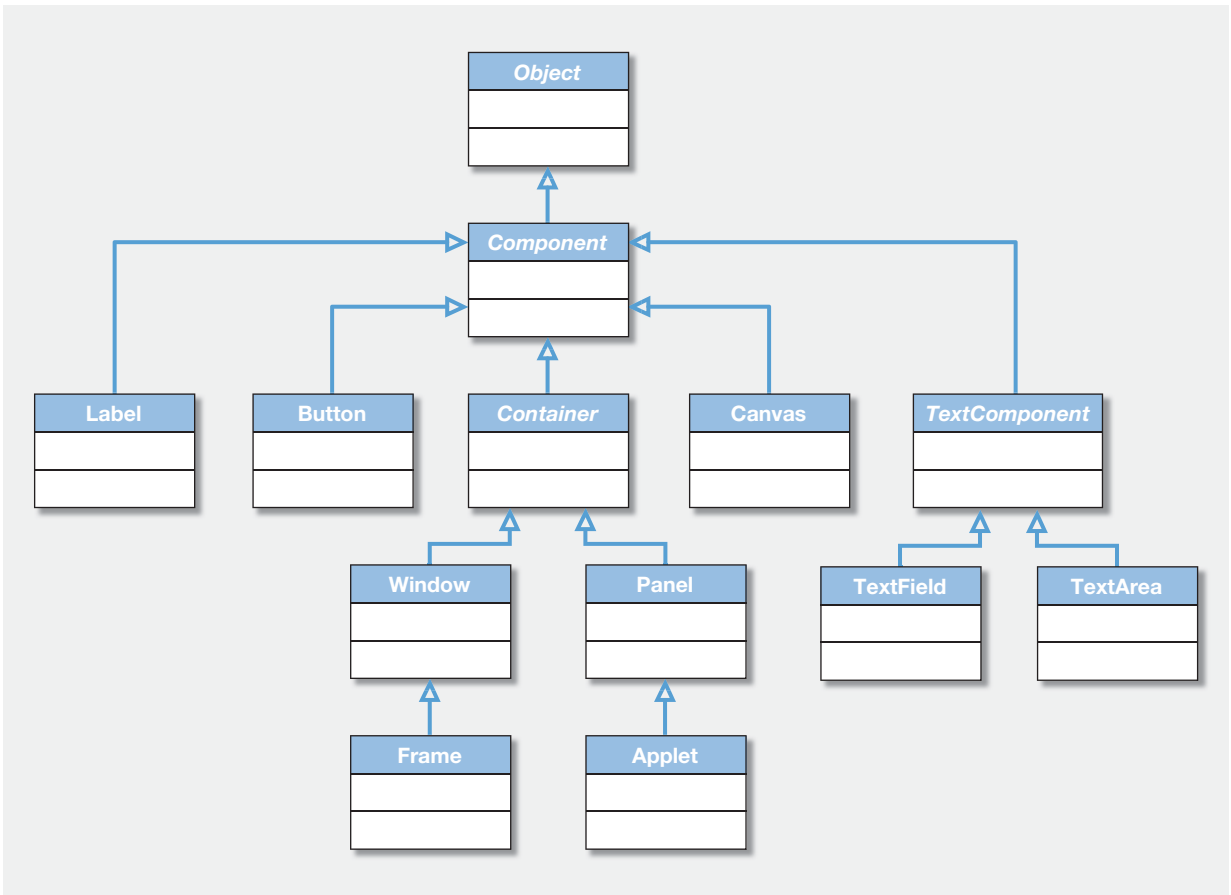


FIGURA **1**

OSSERVAZIONE La gerarchia delle classi della libreria AWT ha come radice la classe astratta *Component*, da cui derivano tutte le classi che implementano concretamente un componente. La classe *Container* è una classe astratta, derivata da *Component*, che rappresenta i contenitori: le sue sottoclassi implementano concretamente i contenitori che vengono utilizzati per realizzare e organizzare un'interfaccia grafica.

1.1 Contenitori

Finestra (*Window*)

La finestra è il tipo di contenitore principale e viene normalmente implementata come istanza della classe *Frame*: ogni applicazione dotata di una GUI deve avere almeno una finestra nella quale posizionare i componenti grafici.
Esistono due diversi costruttori per questa classe: `Frame()` e `Frame(String title)`; entrambi creano una finestra non immediatamente visibile, il primo priva del titolo, il secondo con il titolo fornito come parametro. I metodi `setSize(int width, int height)`, `setLocation(int x, int y)` e `setVisible(boolean visible)` sono utilizzati rispettivamente per definire le dimensioni della finestra in pixel, assegnare una posizione iniziale all'interno dello schermo e a renderla, o meno, visibile.

ESEMPIO

Il frammento di codice Java che segue istanzia un oggetto di tipo finestra avente come titolo «Finestra», di dimensioni 400 × 200 pixel e posizionata alle coordinate (100, 200) dello schermo[1]:

```
…
Frame f = new Frame("Finestra");
f.setSize(400, 200);
f.setVisible(true);
f.setLocation(200, 100);
…
```

1. L'angolo in alto a sinistra della finestra sarà posizionato a questa coordinata espressa in pixel calcolata a partire dall'angolo in alto a sinistra dello schermo.

Pannello (*Panel*)

La classe *Panel* della libreria AWT implementa un tipo di contenitore usato principalmente per raggruppare i componenti grafici e posizionarli in un altro contenitore (*Frame* o *Panel*).

ESEMPIO

Il frammento di codice Java che segue istanzia due pulsanti (*Button*) e un pannello in cui essi sono posizionati invocando il metodo *add*; il pannello viene a sua volta posizionato all'interno di una finestra analoga a quella dell'esempio precedente:

```
…
Button b1 = new Button("SI");
Button b2 = new Button("NO");
```

```
Panel p = new Panel();
p.add(b1);
p.add(b2);

Frame f = new Frame("Finestra");
f.setSize(400, 200);
f.add(p);
f.setVisible(true);
f.setLocation(200,100);
…
```

1.2 Componenti

Etichette (*Label*)

Le etichette sono i componenti grafici più semplici: sono istanze della classe *Label* e possono contenere una sola riga di testo.

ESEMPIO

Un'etichetta, nella sua forma più semplice, può essere istanziata con un'istruzione come la seguente:

```
Label l = new Label("Etichetta");
```

Una volta istanziata un'etichetta, come un qualsiasi altro componente grafico, può essere inserita in un contenitore utilizzando il metodo *add*.

Caselle di testo (*Text field*)

La classe *TextField* di AWT permette di creare caselle di testo: *TextField* deriva dalla classe *TextComponent* e i suoi oggetti visualizzano caselle di una sola riga di testo utilizzate per l'input/output di stringhe di caratteri.

ESEMPIO

Una casella di testo, nella sua forma più semplice, può essere istanziata con la seguente istruzione:

```
TextField t = new TextField("Testo", 10);
```

dove «Testo» è la stringa visualizzata inizialmente e 10 il numero massimo di caratteri inseribili.

Le caselle di testo consentono all'utente di inserire, modificare e visualizzare dati; i due metodi fondamentali sono `setText(String text)`, che visualizza la stringa fornita come parametro, e `String getText()`, che recupera la stringa presente nella casella.

Pulsanti (*Button*)

I pulsanti sono gli elementi grafici utilizzati per invocare un'azione quando vengono selezionati e premuti mediante il mouse. Nella libreria AWT sono istanze della classe *Button*. Un pulsante solitamente comprende una stringa visualizzata sopra di esso che permette all'utente di associarvi la corretta funzionalità.

ESEMPIO

Nella sua forma più semplice un pulsante può essere istanziato con un'istruzione come questa:

```
Button b = new Button ("OK");
```

1.3 Disposizione dei componenti grafici

L'inserimento dei componenti grafici in un contenitore necessita di un criterio che permetta di posizionarli in modo tale che l'interfaccia per l'utente risulti usabile e razionale. Relativamente ai componenti grafici AWT, il loro posizionamento all'interno di un contenitore dipende da un oggetto «gestore dell'aspetto» che organizza la disposizione dei vari elementi. Il gestore è denominato ***Layout Manager*** e i contenitori ne hanno uno di default che può essere eventualmente modificato dal programmatore invocando il metodo *setLayout*.

ESEMPIO

Per assegnare al pannello *p* il gestore di aspetto *BorderLayout* è sufficiente la seguente riga di codice:

```
p.setLayout(new BorderLayout());
```

I principali tipi di *layout manager* previsti dalla libreria AWT sono: *FlowLayout*, *BorderLayout* e *GridLayout*.

FlowLayout

Il gestore di tipo *FlowLayout* viene usato per ordinare i componenti grafici in riga, uno di seguito all'altro; quando una riga è completa si passa a quella successiva. I componenti sono centrati nella riga e sono separati da un piccolo spazio. È il gestore di default dei pannelli.

BorderLayout

Questo tipo di gestore prevede la suddivisione del contenitore in cinque aree denominate *North*, *South*, *Center*, *East* e *West* come mostrato nella FIGURA **2**.

Il gestore massimizza lo spazio assegnato all'area centrale, lasciando alle altre aree lo spazio strettamente necessario per la visualizzazione del loro contenuto. È possibile inserire componenti grafiche solo in alcune aree,

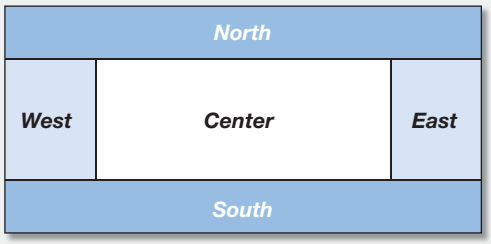


FIGURA **2**

lasciando le altre vuote: in questo caso lo spazio viene utilizzato dai soli componenti grafici presenti. È il gestore di default delle finestre.

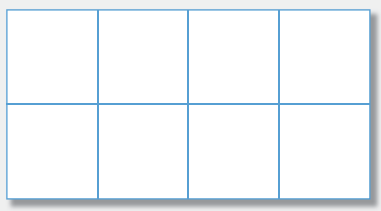
GridLayout

Questo tipo di gestore permette di suddividere un contenitore in aree uguali secondo una griglia formata da righe e colonne: ogni singolo componente grafico può occupare una sola cella della griglia. Il costruttore principale prevede come parametri il numero di righe e il numero di colonne della griglia, o – in alternativa – il numero di righe e il numero di colonne più la spaziatura espressa in pixel tre le righe e tra le colonne.

ESEMPIO

Il seguente frammento di codice istanzia un oggetto di classe *Frame* a cui viene associato un gestore dell'aspetto che realizza la griglia illustrata:

```
Frame f = new Frame("Finestra");
f.setLayout(new GridLayout(2, 4));
```



I componenti grafici vengono inseriti invocando il metodo *add*: essi sono posizionati nell'ordine di riga/colonna in cui sono aggiunti.

OSSERVAZIONE Impostando a `null` il *layout manager* di un contenitore è possibile gestire il posizionamento assoluto dei singoli componenti grafici: questa tecnica di codifica risulta però estremamente complessa ed è di fatto impiegata esclusivamente degli strumenti di programmazione visuale degli ambienti integrati di sviluppo.

1.4 La gestione degli eventi

L'interazione dell'utente con gli elementi di un'interfaccia grafica si suddivide in due azioni fondamentali:

- intercettare le azioni compiute dall'utente (compito affidato al sistema di gestione degli eventi);
- associare alle azioni dell'utente l'invocazione di specifici metodi da eseguire come risposta agli eventi intercettati (compito affidato allo sviluppatore dell'applicazione).

In pratica gli eventi sono una modalità di comunicazione tra l'utente e il programma in esecuzione. I principali tipi di eventi possono essere così classificati:

- **input da parte dell'utente**:
  - pressione di un pulsante del mouse,
  - pressione di un tasto della tastiera,
  - …
- **eventi dell'interfaccia utente**:
  - click su pulsanti o altri componenti grafici,
  - apertura di un menu,
  - …
- **eventi delle finestre**:
  - apertura o chiusura di una finestra,
  - miniaturizzazione o ripristino di una finestra,
  - …

Per garantire una rapida risposta alle azioni dell'utente il linguaggio di programmazione Java basa la propria gestione degli eventi su un modello a **delegazione**, implementato da uno o più **ascoltatori** di eventi (***listener***) responsabili della gestione di eventi specifici.
Per la gestione degli eventi è quindi necessario coinvolgere due oggetti distinti:

- l'oggetto che origina l'evento (un componente grafico della GUI);
- l'oggetto ascoltatore di eventi, che può essere diverso per categorie di eventi distinte, che invoca a sua volta i metodi in risposta agli eventi specifici che deve gestire.

I due oggetti sono tra loro collegati mediante la **registrazione** dell'ascoltatore su uno o più componenti della GUI che si intendono controllare, limitatamente alle tipologie di eventi di interesse.

OSSERVAZIONE In realtà esiste un terzo oggetto: l'evento stesso, che viene fornito come parametro all'ascoltatore. In molti casi si è interessati al solo fatto che l'evento si sia verificato attivando il *listener*, per cui l'oggetto evento non risulta di fatto utile.

Per gestire gli eventi della GUI è in pratica necessario eseguire le seguenti operazioni:

- individuare gli eventi da gestire e creare uno o più ascoltatori a partire dalle specifiche interfacce definite nella libreria AWT;
- definire i metodi per la gestione degli eventi associati agli ascoltatori;
- registrare gli ascoltatori su uno o più oggetti di origine degli eventi che si intendono controllare (tipicamente un componente grafico della GUI).

La libreria AWT rende disponibili molte interfacce di tipo *Listener* per la realizzazione di classi adatte a intercettare e gestire le più svariate tipologie di eventi generati dai componenti di una GUI; nella TABELLA **1** sono riportate solo quelle di uso più frequente.

TABELLA **1**

Evento	Interfaccia *Listener*	Metodo di registrazione	Metodo/i fondamentale/i dell'interfaccia	Descrizione dell'evento
ActionEvent	*ActionListener*	*addActionListener*	*actionPerformed*	Azione fondamentale di un componente (per esempio il click di un pulsante)
KeyEvent	*KeyListener*	*addKeyListener*	*keyPressed* *keyReleased* *keyTyped*	Tasto premuto, rilasciato o digitato
MouseEvent	*MouseListener*	*addMouseListener*	*mouseClicked* *mouseEntered* *mouseExited* *mousePressed* *mouseReleased*	Pulsante del mouse attivato, premuto o rilasciato; entrata/uscita del cursore del mouse dal componente
TextEvent	*TextListener*	*addTextListener*	*textValueChanged*	Testo modificato
WindowEvent	*WindowListener*	*addWindowListener*	*windowActivated* *windowClosed* *windowClosing* *windowDeactivated* *windowDeiconifed* *windowIconifed* *windowOpened*	Finestra attivata/disattivata, aperta/chiusa, miniaturizzata/ripristinata

Tutti i contenitori e i componenti della libreria AWT comprendono i metodi ***add…Listener*** per la registrazione degli ascoltatori di specifici eventi: ogni componente permette di registrare solo ascoltatori coerenti con la propria funzione.

ESEMPIO

Per intercettare l'evento di pressione di un pulsante, una classe deve implementare l'interfaccia *ActionListener* e definire il metodo *ActionPerformed* che sarà invocato ogni volta che l'utente effettuerà un «click» sul pulsante:

```
public class GUI extends Frame implements ActionListener {
   …
```

```
    private Button b = new Button("Pulsante");
    …
    public GUI() {
       …
       this.add(b);
       b.addActionListener(this);
       …
    }

    …
    …
    …

    public void actionPerformed(ActionEvent event) {
       …
       …
       …
    }
}
```

Dato che un gestore di eventi è una classe che implementa una delle interfacce di tipo *Listener*, si hanno due diverse possibilità:

- creazione di una specifica classe per la gestione degli eventi;
- gestione degli eventi direttamente nella classe che realizza l'interfaccia grafica.

OSSERVAZIONE La libreria AWT rende disponibili alcune classi astratte – denominate **adattatori** (***Adapter***) – che implementano le interfacce di tipo *Listener* e che permettono di derivare classi utilizzabili per istanziare un oggetto ascoltatore; le classi *Adapter* più utilizzate sono le seguenti:

- *KeyAdapter*,
- *MouseAdapter*,
- *WindowsAdapter*.

Queste classi realizzano le funzionalità di un ascoltatore di più eventi correlati, consentendo la gestione completa di un aspetto dell'interazione con l'utente (tastiera, mouse, finestra, …).

ESEMPIO

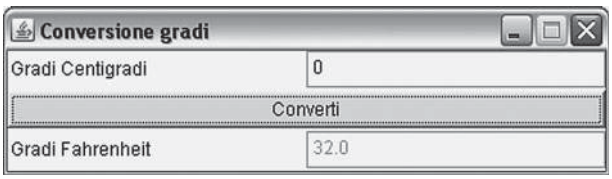
La classe che segue implementa un convertitore da gradi centigradi a gradi Fahrenheit dotato di una semplice GUI realizzata con contenitori e componenti della libreria AWT:

```
// package AWT
import java.awt.*;
import java.awt.event.*;

public class Convertitore extends Frame implements ActionListener, WindowListener {
```

```
// casella di testo per dato (input)
private TextField centigradi = new TextField("", 40);
// casella di testo per risultato (output)
private TextField fahrenheit = new TextField("", 40);
// pulsante per l'azione di conversione
private Button converti = new Button("Converti");

// costruttore
public Convertitore() {
  super("Conversione gradi");   // titolo finestra
  setSize(400, 110);            // dimensione finestra
  setResizable(false);          // finestra non ridimensionabile
  fahrenheit.setEnabled(false); // casella di testo non editabile
  setVisible(true);             // finestra visibile
  setLocation(200,150);         // posizione iniziale finestra
  // disposizione contenitori e componenti grafici
  this.setLayout(new GridLayout(3, 1, 2, 2));
  Panel p1 = new Panel();
  p1.setLayout(new GridLayout(1,2));
  Panel p2 = new Panel();
  p2.setLayout(new GridLayout(1,2));
  p1.add(new Label("Gradi Centigradi"));
  p2.add(new Label("Gradi Fahrenheit"));
  p1.add(centigradi);           // casella di testo per dato
  this.add(p1);
  this.add(converti);           // pulsante per la conversione
  this.add(p2);
  p2.add(fahrenheit);           // casella di testo per risultato
  // registrazione gestori di eventi finestra e pulsante
  addWindowListener(this);
  converti.addActionListener(this);
}

// metodo invocato dall'evento di click sul pulsante di azione
public void actionPerformed(ActionEvent event) {
  double gradiCentigradi, gradiFahrenheit;

  try  {
        // acquisizione e conversione in formato numerico dato di input
        String valore = centigradi.getText();
        gradiCentigradi = Double.parseDouble(valore);
        // calcolo risultato conversione
        gradiFahrenheit = 32.0 + gradiCentigradi*1.8;
        // conversione in stringa e visualizzazione del risultato
        fahrenheit.setText(Double.toString(gFahrenheit));
      }
  catch(Exception exception) {
    fahrenheit.setText("");
  }
}

// metodi listener eventi finestra
public void windowIconified(WindowEvent event) {}
```

```
    public void windowDeiconified(WindowEvent event) {}
    public void windowActivated(WindowEvent event) {}
    public void windowDeactivated(WindowEvent event) {}
    public void windowOpened(WindowEvent event) {}
    public void windowClosed(WindowEvent event) {}
    public void windowClosing(WindowEvent event) {
      System.exit(0); // uscita dal programma
    }

    public static void main(String args[]){
      Convertitore convertitore = new Convertitore();
    }
}
```

L'esecuzione del *main* della classe *Convertitore* crea una semplice GUI di interazione con l'utente:



che consente l'inserimento del valore di temperatura espresso in gradi centigradi e, effettuando un «click» sul pulsante «Converti», di ottenere il valore corrispondente espresso in gradi Fahrenheit.

OSSERVAZIONE Analizzando il codice dell'esempio precedente si nota che:

- sono stati importati i package AWT per la gestione dei contenitori e componenti grafici (`java.awt.*`) e degli eventi (`java.awt.event.*`);
- la classe *Convertitore* che implementa l'applicazione realizza una finestra perché estende direttamente la classe *Frame* della libreria AWT;
- la stessa classe *Convertitore* gestisce direttamente gli ascoltatori degli eventi della finestra e del pulsante implementando rispettivamente le interfacce *WindowListener* e *ActionListener*;
- sono stati impiegati 5 componenti grafici: le 2 etichette anonime per le stringhe «Gradi Centigradi» e «Gradi Fahrenheit», le 2 caselle di testo *centigradi* e *fahrenheit* (quest'ultima, essendo usata solo per visualizzare il risultato della conversione è stata resa non editabile invocando il metodo `setEnabled(false)`) e il pulsante *converti*;
- il costruttore provvede alla registrazione degli ascoltatori per gli eventi della finestra dell'applicazione (`addWindowListener(this)`) e per il pulsante (`converti.addActionListener(this)`);
- sono istanziati 2 pannelli *p1* e *p2* che, organizzati con gestore dell'aspetto di tipo *GridLayout*; contengono rispettivamente un'etichetta e una casella di testo ciascuno (*p1* per i gradi centigradi e *p2* per i gradi Fahrenheit).

- anche alla finestra è stato applicato un gestore dell'aspetto di tipo *GridLayout* con griglia di tre righe e una colonna per l'inserimento, nell'ordine, del pannello *p1*, del pulsante *converti* e del pannello *p2*;
- l'implementazione del metodo *actionPerformed* definito dall'interfaccia *ActionListener* realizza l'ascoltatore per l'evento fondamentale del pulsante *converti*: quando si effettua un «click» il metodo viene invocato per effettuare la conversione dei gradi da centigradi a Fahrenheit;
- i metodi il cui nome inizia con il prefisso *window*… implementano gli ascoltatori degli eventi della finestra dell'applicazione: essendo definiti dall'interfaccia *WindowListener* devono essere dichiarati anche se non si ha interesse alla loro gestione; il metodo *windowClosing* deve invece essere effettivamente implementato per garantire la terminazione del programma quando si attiva il pulsate di chiusura della finestra dell'applicazione.

OSSERVAZIONE Nell'esempio precedente la gestione degli eventi generati dall'interazione dell'utente con la finestra è stata implementata nella classe stessa che realizza l'interfaccia grafica. Una possibile alternativa è quella di definire una classe derivata da *WindowsAdapter* e utilizzare come ascoltatore un oggetto istanza di questa classe:

```
  import java.awt.event.*;

  class GestoreFinestra extends WindowAdapter {
    public void windowClosing(WindowEvent event){
      System.exit(0); // uscita dal programma
    }
}
```

In questo caso la classe *Convertitore* non deve implementare l'interfaccia *WindowListener* e per la registrazione dell'ascoltatore nel costruttore si invoca

```
addWindowListener(new GestoreFinestra())
```

anziché

```
addWindowListener(this);
```

Anche senza estendere una classe *Adapter* predefinita, è possibile definire come ascoltatore una classe diversa da quella che realizza la GUI. Il codice che segue esemplifica questo approccio nel caso del pulsante del convertitore:

```
import java.awt.*;
import java.awt.event.*;

class GestorePulsante implements ActionListener {
  private TextField centigradi;
  private TextField fahrenheit;
```

```
  public GestorePulsante(TextField centigradi,
                         TextField fahrenheit) {
    this.centigradi = centigradi;
    this.fahrenheit = fahrenheit;
  }

  public void actionPerformed(ActionEvent event) {
    double gradiCentigradi, gradiFahrenheit;

    try {
          // acquisizione e conversione in formato numerico
          // dato di input
          String valore = centigradi.getText();
          gradiCentigradi = Double.parseDouble(valore);
          // calcolo risultato conversione
          gradiFahrenheit = 32.0 + gradiCentigradi*1.8;
          // conversione in stringa e visualizzazione
          // del risultato
          fahrenheit.setText(Double.toString
                             (gradiFahrenheit));
        }
    catch(Exception exception) {
      fahrenheit.setText("");
    }
  }
}
```

Anche in questo caso la classe *Convertitore* non deve implementare l'interfaccia *ActionListener* e per la registrazione dell'ascoltatore si invoca

```
converti.addActionListener(new
            GestorePulsante(centigradi, fahrenheit));
```

invece di

```
            converti.addActionListener(this);
```

1.5 La aree di disegno

Tutti i componenti grafici introdotti fin qui sono caratterizzati da un aspetto predefinito; è tuttavia possibile disegnare forme geometriche all'interno dell'area visibile del componente *Canvas*: questa classe definisce il metodo *paint*, che viene invocato automaticamente – in modo analogo a un ascoltatore di eventi – ogni volta che l'area del componente viene visualizzata o modificata. L'uso tipico della classe *Canvas* consiste nel derivare una classe che ridefinisce il metodo *paint* utilizzando gli «strumenti» resi disponibili dall'oggetto di tipo *Graphics* che esso riceve come parametro per disegnare forme geometriche (linee, poligoni, …) nell'area visibile del componente:

```
import java.awt.*;

public class Disegno extends Canvas {
  public void paint (Graphics graphic_context) {
    …
    // disegno di forme geometriche
    …
  }
}
```

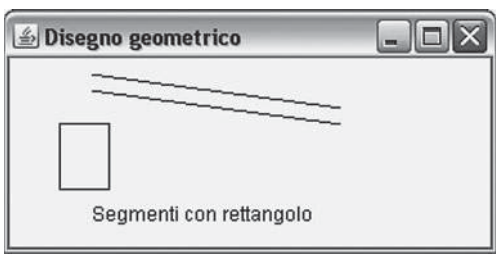
Il parametro del metodo *paint* è normalmente un'oggetto istanza della classe *Graphics2D* derivata dalla classe astratta *Graphics*: esso rappresenta il contesto grafico di riferimento per il componente e in particolare i metodi *draw…* e *fill…* consentono di disegnare forme geometriche nell'area di un componente di tipo *Canvas*.

ESEMPIO

Il seguente codice Java

```
import java.awt.*;

public class Disegno extends Canvas {
  public void paint (Graphics graphic_context) {
    graphic_context.drawLine(50, 10, 200, 30);
    graphic_context.drawLine(50, 20, 200, 40);
    graphic_context.drawRect(30, 40, 30, 40);
    graphic_context.drawString("Segmenti con rettangolo", 50, 100);
  }
}

class Finestra {
  public static void main(String args[]) {
    Frame f = new Frame("Disegno geometrico");
    f.setSize(300, 150);
    Disegno d = new Disegno();
    f.add(d);
    f.setVisible(true);
  }
}
```

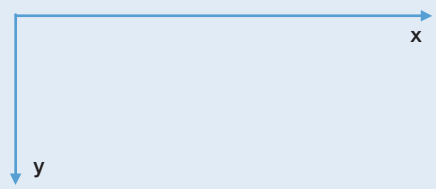
visualizza la seguente finestra



OSSERVAZIONE Analizzando il codice dell'esempio precedente si nota che:

- il metodo *paint* è invocato automaticamente al momento della costruzione dell'oggetto *d* di classe disegno che viene successivamente aggiunto al frame *f* della finestra;
- i due segmenti visualizzati sono stati tracciati invocando il metodo *drawLine* (i cui parametri specificano le coordinate dei vertici); il rettangolo è tracciato con il metodo *drawRect* (in questo caso i parametri sono le coordinate del vertice in alto a sinistra, la larghezza e l'altezza del rettangolo); infine la stringa «Segmenti con rettangolo» è tracciata dal metodo *drawString* (i tre parametri specificano il testo da visualizzare e le coordinate del punto a partire dal quale viene disegnata la stringa);
- la mancata implementazione dell'interfaccia *WindowListner* impedisce la chiusura della finestra dell'applicazione che deve essere forzata.

Il sistema di coordinate a cui i metodi invocati fanno riferimento ha origine nell'angolo in alto a sinistra del componente grafico e i suoi assi sono così orientati:



2 Il *pattern* architetturale *Model-View-Control* e la separazione tra logica di *business* e GUI

Molte librerie di componenti grafici e molte applicazioni che interagiscono con l'utente mediante una GUI realizzano un *pattern* architetturale denominato MVC (*Model-View-Controller*) che prevede la separazione dell'implementazione dei seguenti elementi:

- visualizzazione grafica (*view*);
- rappresentazione della logica applicativa (*model*);
- gestione dell'interazione con l'utente (*controller*).

OSSERVAZIONE La «logica applicativa» – spesso denominata «logica di *business*» – è l'insieme delle classi che implementano le «operazioni» di un'applicazione software, indipendentemente dalla loro gestione da parte dell'utente.

Applet Java

Le *applet* sono programmi codificati in linguaggio Java che possono essere incorporati in pagine HTML. Quando un browser abilitato all'uso della tecnologia Java visualizza una pagina web che contiene una *applet*, il codice viene trasferito insieme al contenuto della pagina ed eseguito dalla JVM del computer su cui è attivo il browser.

Le *applet* sono programmi Java dotati di una GUI che non possono funzionare autonomamente, ma solo nel contesto di una pagina web visualizzata da browser.

Anche se ormai in disuso perché sostituite da tecnologie più evolute, le *applet* hanno conosciuto, nei primi anni di vita del linguaggio Java, una grande diffusione legata allo sviluppo della rete Internet.

Per prevenire eventuali problematiche di sicurezza, le *applet* sono soggette a una serie di limitazioni operative aventi lo scopo di evitare che un anonimo programma scaricato dalla rete ed eseguito su un computer possa eseguire operazioni potenzialmente dannose.

ESEMPIO

In un componente casella di testo, l'elemento *view* ha il compito di visualizzare graficamente il componente stesso e il suo contenuto testuale (dimensioni, font dei caratteri, colore, …), l'elemento *model* si occupa del mantenimento del testo digitato che rappresenta il dato da elaborare e l'elemento *controller* deve gestire le interazioni che l'utente effettua con la casella di testo utilizzando la tastiera e il mouse (digitazione dei singoli caratteri, selezione mediante il cursore del mouse, …).

Nei componenti della libreria AWT del linguaggio Java il *pattern* MVC è realizzato in un modo semplificato: è infatti previsto un unico elemento per l'implementazione degli aspetti *view* e *controller*.

Lo schema funzionale[2] di FIGURA **3** illustra il comportamento dei vari elementi previsti dal *pattern* MVC.
Il ricorso al *pattern* MVC consente di suddividere la complessità di realizzazione di un'applicazione dotata di GUI allo scopo di rendere più semplici ed efficaci lo sviluppo, la manutenzione e la riusabilità del codice.

2. S. Rossini, L. Dozio, *Il pattern MVC*, MokaByte n. 70, Gennaio 2003 (www.mokabyte.it).

La separazione in classi distinte degli elementi *view* e *control* può non essere la soluzione più pratica nel caso di applicazioni molto semplici: è comunque importante mantenere la classe che realizza la GUI indipendente dalla classe che implementa la logica di *business* dell'applicazione.

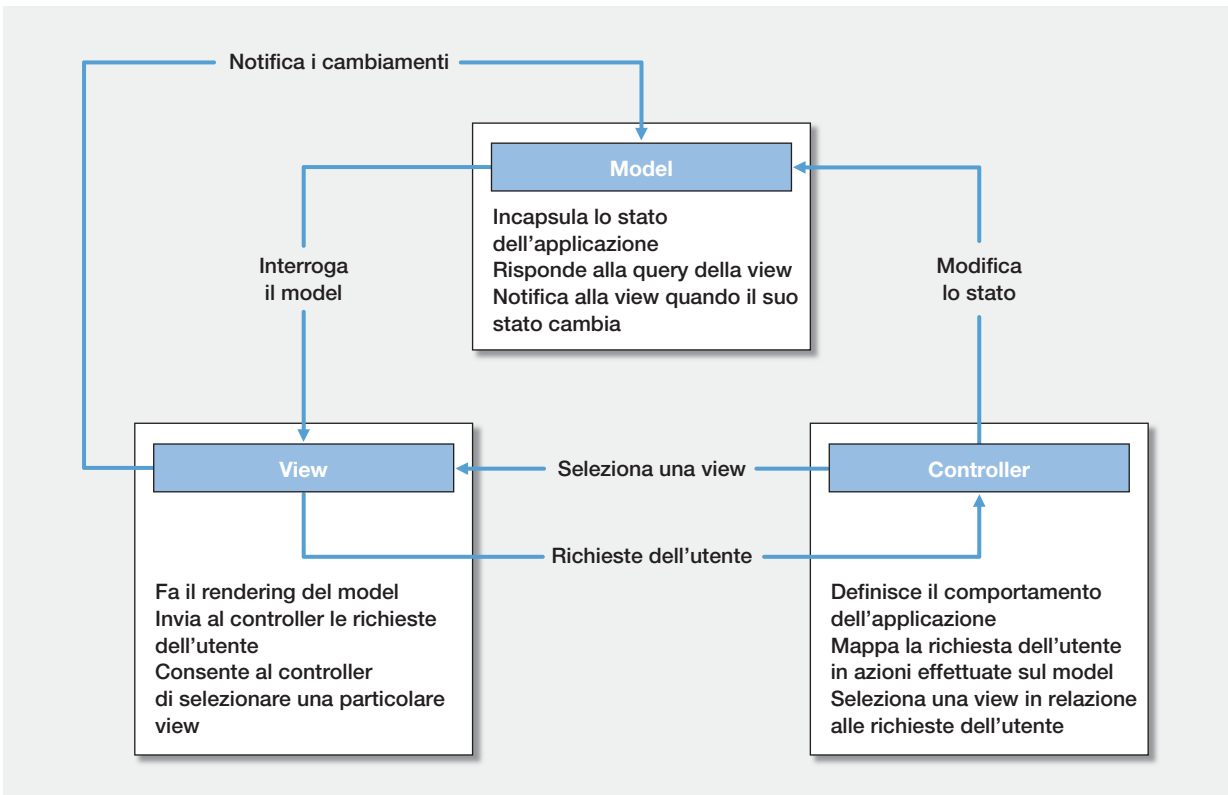


FIGURA **3**

ESEMPIO

La classe *Calcolatrice* introdotta in apertura del capitolo rappresenta la logica di *business* della semplice calcolatrice grafica implementata dalla seguente classe Java che gestisce sia le funzioni dell'elemento *view* (visualizzazione e aggiornamento dei componenti grafici) sia quelle dell'elemento *controller* (gestione degli eventi generati dall'interazione con l'utente):

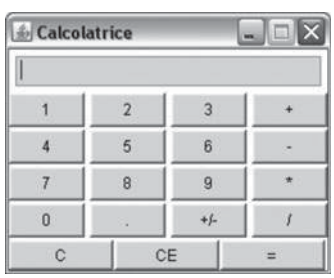
```
import java.awt.*;
import java.awt.event.*;

// classe per la gestione semplificata degli eventi di una finestra
class AscoltatoreEventiFinestra extends WindowAdapter {
  public void windowClosing(WindowEvent event) {
    System.exit(0); // uscita dal programma
  }
}

public class CalcolatriceGUI extends Frame implements ActionListener {
  static final int NUMERO_MASSIMO_CIFRE = 20;
  private Calcolatrice calcolatrice;
  private boolean decimali; // flag presenza separatore decimale
  private boolean nuovoNumero;  // flag nuovo numero inserito
  private TextField display = new TextField(30);
  private Button[] pulsanti = new Button[19];
  private String etichetteTasti = "1,2,3,+,4,5,6,-,7,8,9,*,0,.,+/-,/";
  private String[] tasti = etichetteTasti.split(",");

  // costruttore
  public CalcolatriceGUI() {
    super("Calcolatrice"); // titolo finestra
    // registra l'ascoltatore degli eventi della finestra
    addWindowListener(new AscoltatoreEventiFinestra());
    setSize(250,200); // dimensionamento finestra
    setResizable(false);
    setVisible(true);
    setLocation(200,150); // impostazione posizione iniziale
    setLayout(new BorderLayout()); // gestore aspetto finestra
    // creazione pulsanti
    for (int i=0; i<16; i++) {
      pulsanti[i] = new Button(tasti[i]);
    }
    pulsanti[16] = new Button("C");
    pulsanti[17] = new Button("CE");
    pulsanti[18] = new Button("=");
    // registrazione ascoltatore eventi dei pulsanti
    for (int i=0; i<19; i++) {
      pulsanti[i].addActionListener(this);
    }
    Panel superiore = new Panel(); // pannello per display
    superiore.add(display);
    display.setEditable(false); // disabilitazione inserimento diretto
    Panel centrale = new Panel(); // pannello tastiera operativa
    centrale.setLayout(new GridLayout(4, 2, 4, 2));
```

▶

```
    for (int i=0; i<16; i++) {
      centrale.add(pulsanti[i]);
    }
    Panel inferiore = new Panel(); // pannello pulsanti di controllo
    inferiore.setLayout(new GridLayout(1, 0, 1, 3));
    inferiore.add(pulsanti[16]);
    inferiore.add(pulsanti[17]);
    inferiore.add(pulsanti[18]);
    add("North", superiore);
    add("South", inferiore);
    add("Center", centrale);
    // oggetto che realizza la logica di business
    calcolatrice = new Calcolatrice();
    nuovoNumero = true;
    decimali = false;
  }

  // gestione eventi pulsanti
  public void actionPerformed(ActionEvent event)
  {
    String string = event.getActionCommand(); // pulsante che ha generato l'evento

    // cambio segno
    if (string.equals("+/-")) {
      if (!nuovoNumero)
        calcolatrice.cambiaSegno();
      display.setText(Double.toString(calcolatrice.getOperando()));
    }
    // cifra numerica
    else if ((string.equals("1")) || (string.equals("2")) ||
             (string.equals("3")) || (string.equals("4")) ||
             (string.equals("5")) || (string.equals("6")) ||
             (string.equals("7")) || (string.equals("8")) ||
             (string.equals("9")) || (string.equals("0"))) {
        if (display.getText().length() < NUMERO_MASSIMO_CIFRE) {
          if (nuovoNumero) {
            display.setText(string);
            nuovoNumero = false;
          }
          else {
            string = display.getText()+string;
            display.setText(string);
          }
          calcolatrice.setOperando(Double.parseDouble(string));
        }
    }
    // operatore aritmetico
    else if ((string.equals("+")) || (string.equals("-")) ||
             (string.equals("*")) || (string.equals("/"))) {
          calcolatrice.setOperazione(string.charAt(0));
```

```
            nuovoNumero = true;
            decimali = false;
        }
        // operatore uguale
        else if (string.equals("=")) {
            calcolatrice.uguale();
            display.setText(calcolatrice.risultato());
            nuovoNumero = true;
            decimali = false;
            calcolatrice.setOperazione('#');
        }
        // cancellazione ultimo operando inserito
        else if (string.equals("CE")) {
            calcolatrice.cancellaUltimoOperando();
            display.setText("0");
            decimali = false;
            nuovoNumero = true;
        }
        // cancellazione generale
        else if (string.equals("C")) {
            calcolatrice.cancella();
            display.setText("0");
            decimali = false;
            nuovoNumero = true;
          }
        // separatore decimale
        else if (string.equals(".")) {
            if ((!decimali) && (!nuovoNumero)) {
              decimali = true;
              string = display.getText()+".";
              display.setText(string);
            }
        }
    }

    public static void main(String args[]) {
      CalcolatriceGUI c = new CalcolatriceGUI();
    }
}
```

Il metodo *main* istanzia un oggetto di classe *CalcolatriceGUI* attivando la seguente applicazione che realizza l'interfaccia utente grafica della calcolatrice:



OSSERVAZIONE Il codice del metodo *actionPerformed* della classe *CalcolatriceGUI* gestisce eventi diversi generati dall'interazione dell'utente con l'applicazione (input di cifre numeriche, selezione delle operazioni e richiesta del risultato, controllo del display …). Dato che viene registrato lo stesso ascoltatore su tutti i pulsanti della calcolatrice, è necessario distinguere i singoli eventi: il metodo *getActionCommand* dell'oggetto di tipo *ActionEvent* fornito come parametro permette di effettuare questa distinzione, restituendo una stringa descrittiva che, nel caso di un pulsante, è l'etichetta del pulsante stesso.

3 Il *pattern* comportamentale *Observer* e la programmazione *event-driven*

I *pattern* comportamentali forniscono soluzioni per le più ricorrenti tipologie di interazione tra gli oggetti. In particolare il *pattern* ***Observer*** (**Osservatore**) definisce una dipendenza uno a molti fra un soggetto osservato e vari oggetti «osservatori», in modo che se il soggetto osservato modifica il suo stato, a tutti gli oggetti osservatori che si sono esplicitamente registrati viene notificato il cambiamento avvenuto.

Il ricorso al *pattern Observer* nasce dall'esigenza di mantenere un alto livello di consistenza fra classi correlate, ma senza produrre situazioni di forte dipendenza tra la classe del soggetto osservato e quella degli oggetti osservatori.

OSSERVAZIONE Dovendo notificare i cambiamenti di stato del soggetto agli osservatori, viene richiesto agli osservatori di sottoscriversi presso il soggetto che, mantenendo una lista degli osservatori registrati, provvederà a notificare a ciascuno di essi i propri cambiamenti di stato invocando un metodo specifico. A questo scopo viene normalmente definita una classe astratta, che definisce il prototipo del metodo di notifica, da cui la classe degli osservatori deriva: in questo modo la sottoscrizione di un'istanza di un oggetto osservatore presso il soggetto rende disponibile un metodo specifico di notifica da invocare a ogni aggiornamento dello stato. I diagrammi UML di FIGURA **4** illustrano la soluzione prospettata.

Nel contesto della libreria Java AWT il *pattern Observer* viene utilizzato per realizzare gli ascoltatori (*Listener*) degli eventi asincroni generati dai componenti grafici come, per esempio, gli oggetti istanza della classe *Button*.

L'uso intensivo del *pattern Observer* nella realizzazione delle GUI dà luogo a quella che viene tradizionalmente definita **programmazione *event-driven***: l'esecuzione del codice è infatti prevalentemente guidata dagli eventi generati dall'interazione dell'utente con i componenti grafici.

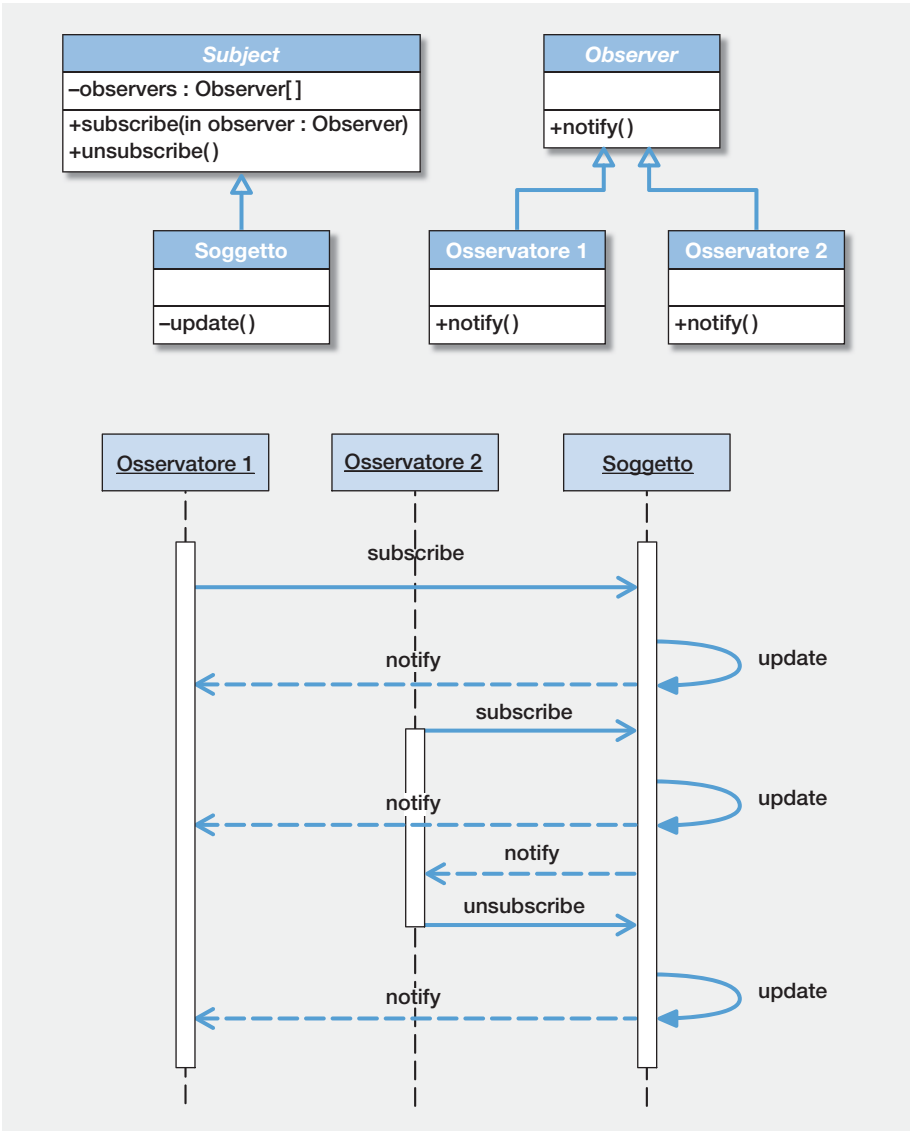


FIGURA **4**

ESEMPIO

La tabella che segue mette in relazione gli elementi del *pattern* con gli elementi della libreria AWT relativi alla gestione dei componenti di tipo *Button*:

Elementi del pattern	Java AWT
Soggetto osservato	Componente *Button*
Osservatore astratto	Interfaccia *ActionListener*
Osservatore concreto	Oggetto che implementa l'interfaccia *ActionListener*

ESEMPIO

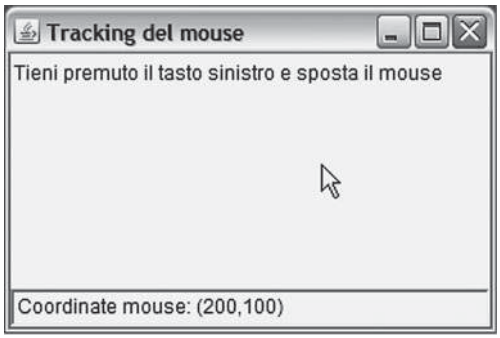
L'applicazione grafica implementata dalla classe Java che segue mostra come realizzare il *tracking* dei movimenti del mouse, sia nel caso in cui un pulsante dello stesso sia premuto (*mouse-dragging*) sia nel caso in cui il mouse sia solo spostato (*mouse-moving*). ▶

Gli spostamenti del mouse sono catturati dai metodi di una classe che implementa l'interfaccia *MouseMotionListener*, ma per catturare gli eventi associati al mouse (come l'effettuazione di un *click*), la stessa classe deve implementare anche l'interfaccia *MouseListener*. Questa soluzione rappresenta una semplice implementazione del *pattern* comportamentale *Observer*: il cambiamento di stato del mouse genera eventi che influenzano lo stato di una casella di testo utilizzata per visualizzarne lo stato corrente.

```
import java.awt.*;
import java.awt.event.*;

class AscoltatoreEventiFinestra extends WindowAdapter {
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
}

public class Mouse implements MouseMotionListener, MouseListener {
  private Frame f;
  private TextField tf;

  public void Mouse() {
    f = new Frame("Tracking del mouse");
    f.addWindowListener(new AscoltatoreEventiFinestra());
    f.add(new Label("Tieni premuto il tasto sinistro e sposta il mouse"),
          BorderLayout.NORTH);
    tf = new TextField(30);
    f.add(tf, BorderLayout.SOUTH);
    f.addMouseMotionListener(this);
    f.addMouseListener(this);
    f.setSize(300,200);
    f.setVisible(true);
  }

  public void mouseDragged(MouseEvent event) {
    String string = "Coordinate mouse: ("+event.getX()+","+event.getY() +")";
    tf.setText(string);
  }

  public void mouseMoved(MouseEvent event) {
  }
  public void mouseClicked(MouseEvent event) {
  }

  public void mouseEntered(MouseEvent event) {
    String s = "Il mouse è nella finestra";
    tf.setText(s);
  }

  public void mouseExited(MouseEvent event) {
    String string = "Il mouse è fuori dalla finestra";
    tf.setText(string);
}
```

```
    public void mousePressed(MouseEvent event) {
    }
    public void mouseReleased(MouseEvent event) {
    }

    public static void main(String args []) {
      Mouse m = new Mouse();
    }
}
```

L'attivazione dell'applicazione GUI mediante invocazione del metodo *mouse* visualizza la seguente GUI:



Sintesi

GUI. Acronimo di ***Graphical User Interface***, usato per indicare il tipo di interfaccia che permette all'utente di un'applicazione software l'interazione con essa mediante elementi grafici piuttosto che comandi testuali.

AWT. L'***Abstract Window Toolkit*** è stato il primo *framework* grafico del linguaggio Java. Esso «mappa» i componenti grafici del sistema operativo ospite con apposite classi *peer* codificate prevalentemente nel linguaggio nativo della piattaforma di esecuzione. Quando un programmatore utilizza un componente grafico AWT nella GUI di un'applicazione Java, viene posizionato sullo schermo un oggetto grafico della piattaforma ospite: le istanze delle classi della libreria AWT provvedono a inviare le invocazioni dei metodi effettuate dall'applicazione sull'oggetto corrispondente della GUI. Allo stesso tempo, quando l'utente dell'applicazione interagisce con un elemento dell'interfaccia grafica, viene creato uno specifico oggetto *Event*, che viene inoltrato al corrispondente componente AWT, in modo tale che il programmatore possa gestirlo in una modalità *object oriented* e in modo completamente indipendente dalla piattaforma di esecuzione. L'interfaccia AWT si basa su due elementi di base: i **componenti** e i **contenitori**.

Componenti. I componenti sono oggetti aventi una rappresentazione grafica. La caratteristica principale di un componente è quella di essere un elemento con il quale l'utente può interagire: esempi di componenti sono i pulsanti (la pressione di un pulsante utilizzando il mouse può avviare un'azione), caselle di testo (la pressione dei tasti dei caratteri in corrispondenza a una casella di testo provoca la visualizzazione dei simboli corrispondenti), ….

Contenitori. I contenitori sono elementi che possono contenere altri componenti. La funzione di un contenitore è quella di permettere il posizionamento e il dimensionamento di compo-

nenti al proprio interno: queste operazioni possono essere eseguite in varie modalità, in funzione del gestore di *layout* (*layout manager*) associato al contenitore stesso. Esempi di contenitori sono le finestre (una finestra è un'area rettangolare con la barra del titolo e i pulsanti di ridimensionamento e chiusura), i pannelli (un pannello è un contenitore che può comprendere al proprio interno vari componenti posizionati, per esempio, all'interno di una finestra o di un diverso pannello), ….

Layout Manager. L'inserimento dei componenti grafici in un contenitore necessita di un criterio che permetta di posizionarli in modo tale che l'interfaccia per l'utente risulti usabile e razionale. Relativamente ai componenti grafici AWT, il loro posizionamento all'interno di un contenitore dipende da un oggetto «gestore dell'aspetto» che organizza la disposizione dei vari elementi. Il gestore è denominato *Layout Manager* e i contenitori ne hanno uno di default che può essere eventualmente modificato dal programmatore invocando il metodo *setLayout*. I tre principali tipi di *Layout Manager* previsti dalla libreria AWT sono: ***FlowLayout***, ***BorderLayout*** e ***GridLayout***.

Programmazione *event-driven*. L'interazione dell'utente con gli elementi di un'interfaccia grafica si suddivide in due azioni fondamentali: intercettare le azioni compiute dall'utente (compito affidato al sistema di gestione degli eventi) e associare alle azioni dell'utente l'invocazione di specifici metodi da eseguire come risposta agli eventi intercettati (compito affidato allo sviluppatore dell'applicazione).

Modello a delegazione. Per garantire una rapida risposta alle azioni dell'utente, il linguaggio di programmazione Java basa la propria gestione degli eventi su un modello «a delegazione» implementato da uno o più **ascoltatori** di eventi (***listener***) responsabili della gestione di eventi specifici. Per la gestione degli eventi è quindi necessario coinvolgere due oggetti distinti: l'oggetto che origina l'evento (un componente grafico della GUI) e l'oggetto ascoltatore di eventi, che può essere diverso per categorie di eventi distinte e che a sua volta invoca i metodi in risposta agli eventi specifici che deve gestire. I due oggetti sono tra loro collegati mediante la «registrazione» dell'ascoltatore su uno o più componenti della GUI che si intendono controllare, limitatamente alle tipologie di eventi di interesse.

WindowListener. È l'interfaccia da implementare per realizzare una classe di tipo *Listener* che riceva gli eventi generati da una finestra. L'oggetto *listener* istanziato da questa classe deve essere registrato su una finestra usando il metodo *addWindowListener*: quando lo stato della finestra cambia (aperta, chiusa, attivata, disattivata, minimizzata o ripristinata), il metodo appropriato della classe viene invocato fornendo come parametro un oggetto di tipo *WindowEvent*.

ActionListener. È l'interfaccia da implementare per realizzare una classe di tipo *Listener* che riceva gli eventi generati da un componente grafico (per esempio un pulsante). L'oggetto *listener* istanziato da questa classe deve essere registrato su una finestra usando il metodo *addActionListener*: quando sul componente viene eseguita un'azione (per esempio il *click* del mouse), il metodo *actionPerformed* della classe viene invocato fornendo come parametro un oggetto di tipo *ActionEvent*.

Canvas. La libreria AWT prevede il componente grafico *Canvas* nella cui area visibile è possibile disegnare forme geometriche. La classe *Canvas* definisce il metodo *paint* che viene invocato automaticamente – in modo analogo a un ascoltatore di eventi – ogni volta che l'area del componente viene visualizzata o modificata. L'uso tipico della classe *Canvas* consiste nel derivare una classe che ridefinisce il metodo *paint* utilizzando gli «strumenti» resi disponibili dall'oggetto di tipo *Graphics* che esso riceve come parametro per disegnare forme geometriche (linee, poligoni, …) nell'area visibile del componente.

MVC. Il *pattern* architetturale **MVC** (***Model-View-Controller***) prevede la separazione dell'implementazione dei seguenti elementi: visualizzazione grafica (***view***), rappresentazione della logica applicativa (***model***) e gestione dell'interazione con l'utente (***controller***). La separazione in classi distinte degli elementi *view* e *control* può non essere la soluzione più pratica nel caso di applicazioni molto semplici: è comunque importante mantenere la classe che realizza la GUI indipendente dalla classe che implementa la logica applicativa.

Observer. Il *pattern* comportamentale *Observer* («Osservatore») definisce una dipendenza uno a molti fra un soggetto osservato e vari oggetti «osservatori», in modo che se il soggetto osservato modifica il suo stato, a tutti gli oggetti osservatori che si sono esplicitamente registrati viene notificato il cambiamento avvenuto. Il ricorso al *pattern Observer* nasce dall'esigenza di mantenere un alto livello di consistenza fra classi correlate, ma senza produrre situazioni di forte dipendenza tra la classe del soggetto osservato e quella degli oggetti osservatori. L'uso intensivo del *pattern Observer* nella realizzazione delle GUI da luogo a quella che viene tradizionalmente definita **programmazione *event-driven***: l'esecuzione del codice è infatti prevalentemente guidata dagli eventi generati dall'interazione dell'utente con i componenti grafici.

QUESITI

1 **Indicare quali delle seguenti affermazioni relative al *framework* AWT sono vere.**

A È l'acronimo di *Abstract Window Toolbox*.
B Può essere utilizzato solo in ambiente Windows.
C Comprende solo classi di tipo *Container*.
D Contiene classi necessarie alla realizzazione di GUI.

2 **Date le seguenti istruzioni**

```
…
Frame f = new Frame("Prova");
f.setSize(200,120);
f.setLayout(new GridLayout(5, 2, 7, 6)
…
```

quali delle seguenti affermazioni sono vere?

A La finestra `f` viene partizionata in 5 righe distanziate tra loro di 2 pixel e 6 colonne distanziate di 7 pixel.
B La finestra `f` viene partizionata in 5 righe distanziate tra loro di 6 pixel e 2 colonne distanziate di 7 pixel.
C La finestra `f` viene partizionata in 2 righe distanziate tra loro di 6 pixel e 5 colonne distanziate di 7 pixel.
D Nessuna delle risposte precedenti.

3 **Indicare quale delle seguenti definizioni corrisponde al concetto di *evento*.**

A Azione asincrona gestita dal sistema allo scadere di un preciso periodo di tempo.
B Azione sincrona generata dall'interazione con l'utente.
C Azione asincrona generata dall'interazione dell'utente con l'applicazione.
D Nessuna delle risposte precedenti.

4 **Per disporre in un contenitore vari componenti grafici in una struttura a tabella è necessario utilizzare …**

A … un gestore di aspetto di tipo *BorderLayout*.
B … un gestore di aspetto di tipo *GridLayout*.
C … un gestore di aspetto di tipo *FlowLayout*.
D … un gestore di aspetto di tipo *TableLayout*.

5 **Le classi AWT che realizzano i componenti e i contenitori sono organizzate in una gerarchia che ha come radice la classe …**

A … *Component*.
B … *Panel*.
C … *Container*.
D … *Root*.

6 **Indicare quali delle seguenti sono attività necessarie per realizzare la gestione degli eventi AWT.**

A Registrare l'ascoltatore su un oggetto origine che si intende controllare.
B Schedulare opportunamente gli eventi.
C Gestire l'evento eseguendo il metodo associato.
D Definire uno o più *listener* per gli eventi che si vogliono gestire.
E Definire una temporizzazione degli eventi.

7 Indicare la forma corretta per definire una classe che gestisce gli eventi generati da un pulsante.

A `class GestorePulsante`

B `class GestorePulsante implements WindowListener`

C `class GestorePulsante implements ActionListener`

D `class GestorePulsante extends ActionListener`

8 Un metodo avente la firma

```
public void actionPerformed(ActionEvent event)
```

può essere utilizzato per prevedere un'azione di risposta all'evento …

A … di apertura di una finestra.

B … di creazione di un nuovo pulsante.

C … di pressione di un pulsante.

D Nessuna delle risposte precedenti.

9 Indicare in quali dei seguenti casi viene applicato il *pattern* comportamentale *Observer*.

A Nella gestione della programmazione *event-driven*.

B Nella costruzione dell'aspetto grafico di una GUI.

C Nell'attività di «osservazione» della realtà al fine di costruirne un modello informatico.

D Nell'implementazione dell'elemento *model* del *pattern* architetturale MVC.

10 Nel *framework* AWT del linguaggio Java il *pattern* MVC viene realizzato separando …

A … gli aspetti *view*, *model* e *control*.

B … l'aspetto *model* da quelli *view* e *control* che sono realizzati mediante un'unica classe.

C … l'aspetto *view* da quelli *model* e *control* che sono realizzati mediante un'unica classe.

D … l'aspetto *control* da quelli *view* e *model* che sono realizzati mediante un'unica classe.

LABORATORIO

1 Realizzare un'applicazione Java con GUI che consenta di convertire una velocità espressa in kilometri/ora in miglia/ora e viceversa (1 km/h = 0,62137 mi/h).

2 Utilizzando gli algoritmi illustrati nel primo volume implementare una classe Java (eventualmente integrata da una o più classi di utilità) che consenta di calcolare:

- il numero di giorni trascorsi tra due date;
- il giorno della settimana corrispondente a una data.

Realizzare un'applicazione Java che consenta all'utente di usufruire delle funzionalità della/e classe/i implementati tramite una GUI.

3 Modificare l'applicazione calcolatrice illustrata nel capitolo in modo da rendere possibile l'inserimento degli operandi direttamente nel display utilizzando la tastiera del computer.

4 Realizzare un'applicazione grafica Java che consenta di «disegnare» rettangoli nell'area visibile di un componente *Canvas* (suggerimento: intercettare gli eventi del mouse per definire uno dei vertici e l'estensione delle dimensioni del rettangolo).

SEZIONE

B

Pagine web con JavaScript

B1 Il linguaggio JavaScript

1 Da applicazioni locali ad applicazioni web

Quasi sempre le applicazioni sono categorizzate in base alla problematica che risolvono. Per esempio si parla applicazioni di tipo *word processor* quando offrono funzionalità avanzate per l'editing e la gestione di documenti.
Fino a poco tempo fa gran parte delle applicazioni di tipo word processor (ma non solo), come Microsoft Word o OpenOffice.org Writer, venivano eseguite sul proprio computer e, al più, gli utenti potevano condividere i documenti generati inviandoli via email. Al giorno d'oggi è possibile eseguire un'applicazione di tipo word processor utilizzando un browser. È il caso di Google Docs, una delle prime applicazioni di tipo word processor fruibile completamente online (di solito di parla anche di applicazioni web). Dagli inizi del 2011 anche la Microsoft ha creato una suite di programmi, denominata Office 365, fruibile interamente via web. In FIGURA 1 l'esempio dell'applicazione Excel di Office 365.
Le applicazioni web, che offrono le stesse funzionalità delle usuali applicazioni eseguite in locale, sono sempre più numerose. Un'applicazione web al

Google Docs

Google Docs permette di creare e gestire documenti, fogli di lavoro e presentazioni. È possibile crearne di nuovi usando template o fare l'upload di propri documenti (sono riconosciuti tutti i principali formati dei più diffusi word processor) e procedere con la loro personalizzazione.
Il vero punto di forza dell'applicazione è la possibilità di permettere l'accesso contemporaneo a più utenti, gestendo una condivisione in tempo reale dei dati e delle loro modifiche.
È anche possibile tenere traccia dei diversi contributi, realizzando così documenti aggiornati, disponibili a tutti o a una cerchia ristretta di collaboratori (estendibile grazie a un sistema a inviti) e senza costi di licenze per l'uso del software di gestione. Si può accedere ai dati contenuti anche da applicazioni esterne grazie a sofisticate API pubbliche.

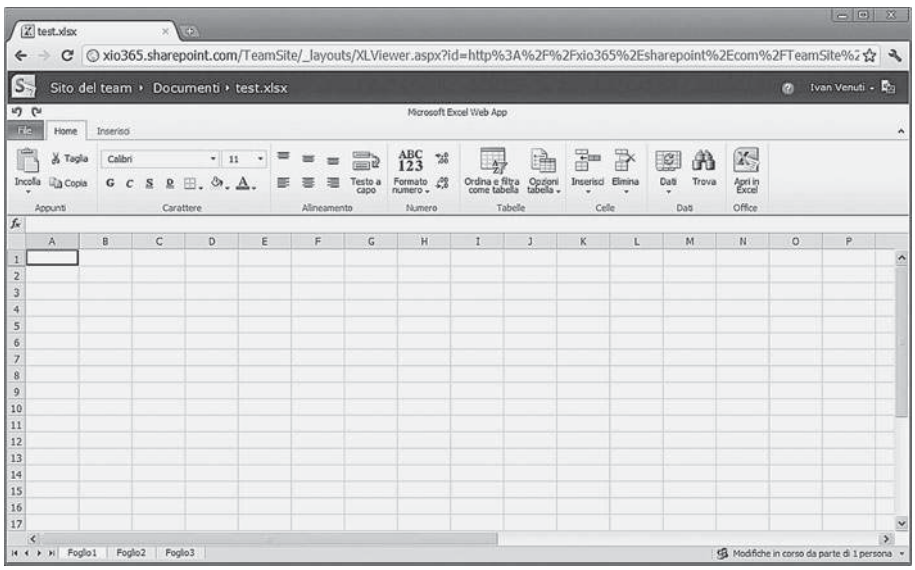


FIGURA 1

giorno d'oggi può avere l'aspetto molto simile alle *applicazioni locali* (chiamate anche *applicazioni stand-alone*, per indicare il fatto che non necessitano di un ulteriore programma, come il browser, per essere eseguite). Oltre alle funzionalità offerte (formattazione dei caratteri, gestione di tabelle, gestione di stili) anche l'aspetto dell'applicazione (finestre multiple, pulsantiere e barre applicative e così via) e, in generale, il *look & feel*, è molto simile tra le applicazioni web e quelle stand-alone. La FIGURA **2** mostra una pagina di GoogleDocs visualizzata a pieno schermo: si potrebbe benissimo pensare a un'applicazione stand-alone.

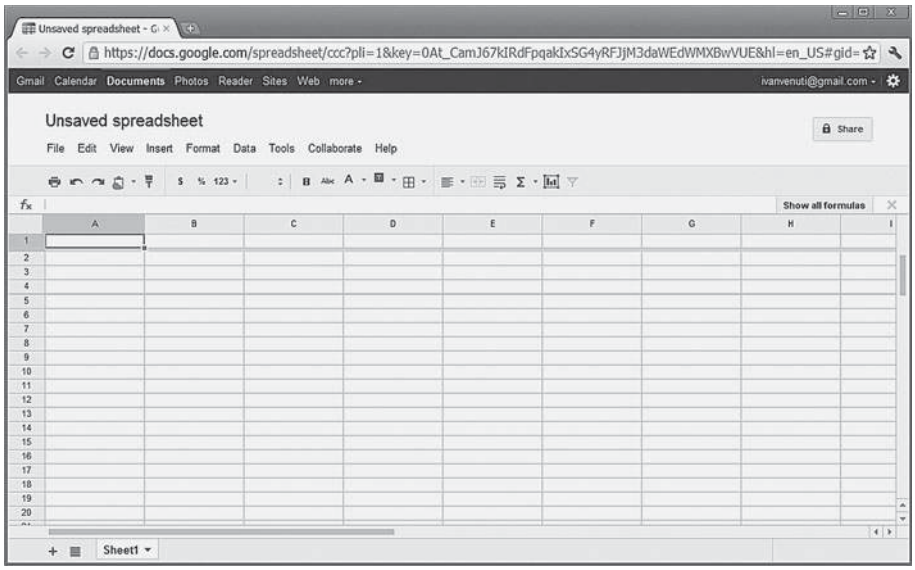


FIGURA **2**

Questa «convergenza» è stata possibile grazie all'evoluzione delle tecnologie web. C'è comunque da domandarsi come mai ci sia stata questa convergenza e quali vantaggi abbiano le applicazioni web.

Un'applicazione web «vive», di fatto, su un server remoto; solo la parte di interfaccia viene creata sul browser, chiamato client. Questo offre alcuni vantaggi per gli utenti: non è necessaria alcuna installazione, c'è una disponibilità immediata di aggiornamenti, il salvataggio dei file può avvenire sia in locale sia sul web e, di conseguenza, i dati sono immediatamente fruibili da altri computer e senza limitazioni (teoriche) di spazio. Chi offre il servizio quasi sempre provvede anche a backup periodici di tutti i dati, minimizzando in questo modo il rischio di perdere informazioni. Inoltre un'applicazione web è indipendente dal sistema operativo, ovvero non necessita, per essere usata, di un sistema operativo particolare (ma, viceversa, c'è possibile dipendenza dal browser utilizzato, come vedremo più avanti).

Queste considerazioni sono valide ponendosi nell'ottica di essere i fruitori delle applicazioni web; ma che dire per quanto riguarda gli *sviluppatori*?

Per realizzare un'applicazione stand-alone è necessario l'utilizzo di un determinato ambiente di programmazione (che comprende la scelta e l'aggiornamento di ambienti di sviluppo, librerie, funzionalità specifiche dei

Look & feel

Il termine *look & feel*, applicato ai programmi, sta a indicare come appare l'interfaccia grafica (*look*) e come si comporta, sia in termini di pulsanti standard sia di icone, pulsanti a scelta rapida e risposta a eventi del mouse quali clic, doppio clic e così via (*feel*).

Per esempio, tutte le suite di programmi hanno il medesimo *look & feel*. Microsoft Office ha lo stesso tipo di pulsantiera (chiamato Ribbon dalla versione 2007). Un sistema operativo ha alcuni comportamenti generali: le finestre di Windows, per esempio, hanno pulsanti per iconizzarle, massimizzarle o chiuderle e una barra delle applicazioni caratteristiche per tale sistema operativo (anche se nelle diverse versioni può subire leggeri cambiamenti).

Dotare un'applicazione web di un *look & feel* appropriato significa permettere che gli oggetti grafici siano visualizzati e si comportino come un qualsiasi altro elemento grafico con cui l'utente è abituato a interagire.

sistemi operativi target). Da tale ambiente va generato un pacchetto di installazione che va distribuito agli utenti.
Per un'applicazione web bisogna scrivere una parte server e una parte client. La parte client è quella visibile sul browser, quella server di solito fornisce gran parte dell'elaborazione e mantiene i dati persistenti. Uno sviluppatore che crea entrambe le parti deve per forza conoscere e usare due tecnologie diverse. In un certo senso c'è un doppio lavoro. Il vantaggio, però, è che la parte client può essere scelta in maniera pressoché indipendente da quella server. Per essere più specifici, un'applicazione che fa uso di HTML e di JavaScript sul client può avere lato server un'applicazione che fa uso di Java, ma anche di un linguaggio della piattaforma .NET o un programma PHP. Questo aspetto ha portato a una situazione attuale in cui gran parte degli sviluppatori sceglie di usare l'HTML e il JavaScript sul client, mentre c'è molta meno uniformità lato server.
In questo capitolo si approfondirà e analizzerà proprio il linguaggio **JavaScript**. Questa conoscenza è, per quanto detto poc'anzi, spendibile in molti contesti lavorativi e diventa un presupposto essenziale per chiunque deciderà di sviluppare siti web professionali.

OSSERVAZIONE Non sono stati citati il linguaggio C né il C++ per lo sviluppo di parti server di un sito web: in linea teorica, qualsiasi linguaggio di programmazione può andar bene per sviluppare qualsiasi progetto, ma di fatto esistono linguaggi dotati di caratteristiche che rendono il loro uso particolarmente indicato per alcune applicazioni o tipologie di programmi. I linguaggi citati (Java, .NET, PHP) sono quelli che, a oggi, sono i più usati (per qualche sito web si utilizzano ancora Perl, ASP o altre tecnologie minori).

2 Programmare il client: oltre l'HTML

L'**HTML** è nato con il World Wide Web. All'inizio c'era l'esigenza di *presentare* pagine con dati ed, eventualmente, di *collegare* tra loro pagine diverse. L'HTML rispondeva bene a questa esigenza, ma non è mai stato un linguaggio di programmazione. Per fare un esempio, una qualunque pagina HTML «pura» è del tutto **s**tatica: l'unico modo per cambiare il contenuto della finestra del browser è quello di caricare un'altra pagina. È questo che avviene quando si fa clic su un link ipertestuale, si preme un pulsante di invio dati di una form o si digita un nuovo indirizzo sulla barra di navigazione.
Nel tempo questo ha costituito, in molti contesti, un grosso limite; per superare tale limite è stato realizzato un nuovo linguaggio di scripting, ovvero un linguaggio che possa «convivere» all'interno di pagine HTML e permettere quella dinamicità che mancava, consentendo la scrittura di veri e propri programmi. Nasce così, nel 1995, JavaScript a opera degli sviluppatori della Netscape, autori di uno dei browser più diffusi in quegli anni, Navigator, che raggiunse una percentuale di utilizzo di quasi l'80%, per poi

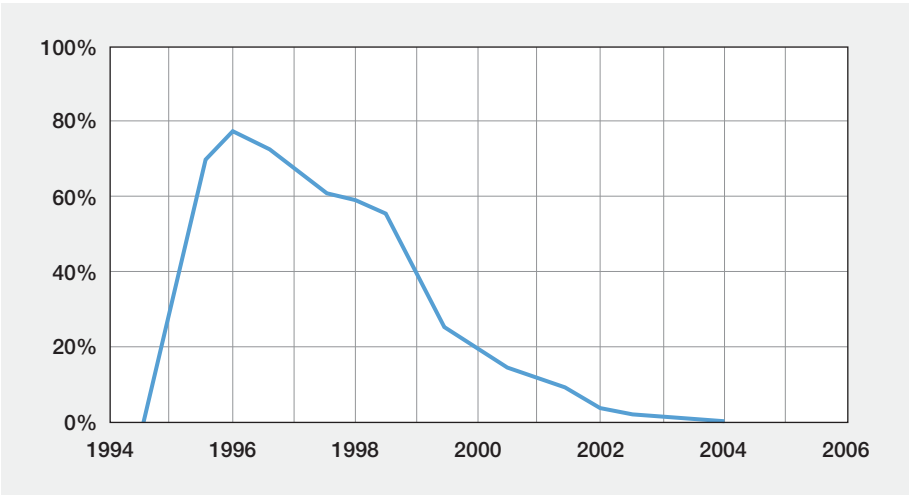


FIGURA 3

scemare, e praticamente scomparire, dopo il 2003, come rappresentato nel grafico[1] di FIGURA 3.

Grazie al nuovo linguaggio JavaScript è possibile, per esempio, intercettare l'evento di apertura o chiusura della pagina, il clic su un pulsante di una form, il cambiamento del contenuto di un campo di inserimento testo, la manipolazione della pagina, aggiungere un campo in più in una form o una riga in più a una colonna e così via.

Per includere una porzione di codice JavaScript dentro una pagina HTML è sufficiente racchiudere il codice tra i tag <script>, in qualunque parte della pagina HTML.

ESEMPIO

Ecco come aprire una piccola finestra per la visualizzazione di un messaggio, la cui esecuzione è mostrata nella FIGURA 4:

```
<script>
  alert("Primo esempio")
</script>
```

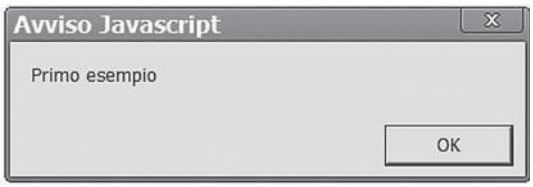


FIGURA 4

La proposta è piaciuta agli utilizzatori, tanto che anche altri produttori, come la Microsoft, si sono presto dotati di funzionalità simili anche per i loro browser.

Purtroppo erano anni in cui ogni produttore cercava di imporre le proprie scelte, per quanto fossero utili e innovative, in autonomia e con caratteri-

1. Fonte Wikimedia: http://commons.wikimedia.org/wiki/File:Netscape-navigator-usage-data.svg.

La «guerra» dei browser

A metà degli anni novanta Netscape produceva uno dei browser più diffusi: Netscape Navigator. Esso veniva usato da circa l'80% di tutti gli utenti delle pagine web. La situazione cambiò radicalmente quando Microsoft decise di puntare molte energie sul proprio browser: Internet Explorer.

Netscape è stata acquisita, nel 1998, da America On Line. Tale acquisizione, anziché dare nuove energie all'azienda, l'ha definitivamente tagliata fuori dal mercato dei browser fino a far scomparire del tutto la loro linea di prodotti browser.

Oggi i browser più diffusi sono Internet Explorer, Mozilla Firefox, Google Chrome, Opera e Safari.

Netscape resta comunque un'azienda importante nella storia del web e nell'evoluzione delle sue tecnologie per aver introdotto innovazioni importanti, quali JavaScript ed SSL (quest'ultima è la tecnologia che permette una comunicazione cifrata tra client e server).

stiche «in più» rispetto alla concorrenza. Questo ha portato ad avere **incompatibilità** più o meno marcate tra browser diversi. Tali incompatibilità erano molto evidenti per il linguaggio di scripting supportato (pertanto il codice JavaScript poteva avere o no certe caratteristiche), ma erano presenti anche nella visualizzazione delle stesse pagine HTML (per esempio mostrando margini diversi per le immagini o le tabelle).
Ad aumentare la confusione si aggiunse il fatto che il nome JavaScript era di proprietà della Netscape; il linguaggio supportato da Internet Explorer, benché molto simile, dovette chiamarsi diversamente e fu deciso di chiamarlo JScript. Non solo: la Microsoft tentò anche la strada della creazione di un linguaggio del tutto diverso, molto simile al suo cavallo di battaglia, il Visual Basic; ecco che Internet Explorer poteva supportare anche il linguaggio di scripting chiamato VBScript. Oramai tale linguaggio è considerato del tutto marginale e praticamente non è più utilizzato.

OSSERVAZIONE Può sembrare strana una competizione sui linguaggi di programmazione fatta a colpi di modifiche fuori standard e con la ricerca di uno spazio di «autonomia» a scapito della compatibilità e dell'interesse comune. In realtà non è un caso tanto isolato; resta famosa una causa intentata da Sun Microsystems, proprietaria di Java, alla Microsoft e al suo utilizzo, difforme dagli standard, di un dialetto di Java per la piattaforma .NET, chiamato J# (la gratella si legge «sharp»). La causa si concluse con un accordo tra le parti ma, di fatto, vide l'uscita di Java dalla piattaforma .NET e dai relativi ambienti di sviluppo.

Avere dei linguaggi simili ma con alcune differenze portò da subito a problemi per gli sviluppatori; non era raro dover sviluppare pagine che «si adattavano», attraverso piccoli escamotage, in base al browser che le visualizzava; in altri casi, per interi siti web c'era la dichiarazione dei browser supportati (anche se spesso si ricorreva alla formula «sito ottimizzato per» e a seguire un particolare tipo di browser).

OSSERVAZIONE Ovviamente la creazione di siti web verificati con un solo browser è una pratica da evitarsi ma ancora molto diffusa. Benché siano passati molti anni, ancora oggi la tentazione di prendere un browser come riferimento e di trascurare gli altri è tutt'oggi forte per molti creatori di siti web. In realtà supportare più di un browser è possibile: basta rispettare gli standard Internet, quelli veri, che non sono decisi da chi realizza i browser ma dal World Wide Web Consortium o W3C. Se un sito si attiene alle specifiche pubbliche del W3C, automaticamente è compatibile con qualsiasi browser.

Nel tempo la situazione è migliorata in quanto le diverse tecnologie (in particolare ci si riferisce all'HTML e a JavaScript) hanno iniziato a essere standard, proposti da uno o più organismi *super partes*, e i singoli browser sempre più cercano di implementarli coerentemente. Purtroppo la situazione non si è mai risolta del tutto, in quanto una vera compatibilità agli

standard era ed è, per ragioni diverse, parziale o addirittura errata, come si avrà modo di analizzare con semplici esempi nel seguito. Al giorno d'oggi ogni nuova versione di browser viene accompagnata da una lista di standard supportati e, per lo meno, con la documentazione delle incompatibilità presenti.
JavaScript resta un nome proprietario di Netscape ma il linguaggio è divenuto uno standard con il nome, usato solo per riferirsi allo standard stesso, di ECMAScript. In questo libro si parlerà di JavaScript intendendo, di fatto ECMAScript e, in particolare, alla versione 5.1 dello standard pubblicata nel giugno del 2011.

OSSERVAZIONE **ECMA International** (http://www.ecma-international.org/) è l'organizzazione a cui è stata affidata la *standardizzazione* del linguaggio JavaScript. In origine l'acronimo ECMA stava per *European Computer Manufacturers Association* poi, dal 1994, poi prese il nome di *Ecma International – European association for standardizing information and communication systems*.

Tutta questa confusione fa sì che, ancora oggi, chi scrive pagine HTML di professione, magari particolarmente complesse sia per il layout sia per gli script utilizzati, deve, per prima cosa, verificare quali versioni di browser è plausibile che vengano usate e far sì che per esse l'applicazione sia testata e pienamente funzionante. Questa scelta rischia di pregiudicare l'uso ottimale per quella parte dei possibili utenti per cui si è deciso di non supportare i loro browser (o, più comunemente, alcune versioni ritenute oramai obsolete).
La scelta di quali browser e quali versioni supportare è quasi sempre dettata dalle statistiche d'uso: se una versione di browser è datata ma ancora molto usata, è auspicabile che si cerchi di rendere compatibili le pagine anche per essa; se, viceversa, è oramai soppiantata da versioni più recenti o è poco usata, probabilmente si può ignorare il fatto che il sito web possa essere visualizzato in maniera non ottimale utilizzando tale versione.

OSSERVAZIONE Un aiuto a realizzare script compatibili con browser diversi viene dall'esistenza di *librerie* di script. Tali librerie si fanno carico delle incompatibilità di basso livello e offrono agli sviluppatori API di più alto livello omogenee per browser di diversi produttori e di diverse versioni. Oltre a tali API di solito tali librerie forniscono anche funzionalità ed effetti grafici avanzati. Ecco perché nel capitolo successivo si prenderà in considerazione una delle librerie più diffuse: **jQuery** (http://jquery.com/).

JavaScript gode oramai di una notevole diffusione e il suo utilizzo avviene nella quasi totalità dei siti e delle applicazioni web. La controprova è molto semplice: aprite la pagina web che preferite (sia essa un sito di giochi, informazione, un blog o altro) e visualizzate il sorgente della pagina. Su tale sorgente ricercate la parola *javascript*. È praticamente certo che la troverete

Web 2.0

Con il termine Web 2.0 si intende un modo diverso di concepire il web dove il concetto chiave, rivoluzionario rispetto al passato, è la *collaborazione*. Grazie a essa compiti complessi o comunque molto costosi, possono essere realizzati grazie a una comunità che si riconosce nei fini e negli obiettivi di un progetto.
Anche progetti «tradizionali» come la realizzazione di un'enciclopedia può basarsi su strategie pre-web, come la creazione di una redazione, la collaborazione di uno o più esperti per singola voce e una gestione centralizzata di tutto il progetto. Un esempio è stata Nupedia.
Oppure può basarsi sulla creazione di una comunità e il compito di chi realizza il sito è quello di fornire gli strumenti di collaborazione, dettare le linee guida per la qualità dei dati e dare delle regole per governare la collaborazione. Un esempio è Wikipedia.
Dei due progetti Nupedia è naufragato nel 2003, mentre Wikipedia è tuttora in grande espansione, con milioni di voci enciclopediche in decine di lingue.

(se non la trovate cercate solo *script*: benché non formalmente corretto, è possibile che compaia solo tale tag senza specificare il linguaggio; potete essere sufficientemente sicuri che il linguaggio usato è proprio JavaScript). Prima di addentrarci nella conoscenza del linguaggio, vediamo alcune applicazioni, sviluppate usando JavaScript come tecnologia principale e non solo come supporto, cercando di capire perché esse si differenzino da siti web scritti utilizzando principalmente l'HTML.

2.1 Pagine il cui contenuto varia dinamicamente: Facebook

Una delle applicazioni più popolari a livello mondiale è **Facebook**. Essa fa parte delle cosiddette applicazioni *social network*, ovvero che permettono la creazione di una comunità virtuale e offrono gli strumenti per gestire aggiornamenti e aumentare la propria rete di contatti. In questo contesto non ci interessa il fenomeno sociale che ha provocato Facebook né vogliamo analizzare le ragioni del suo successo, ma semplicemente ci concentriamo su alcune sue funzionalità rese possibili da JavaScript (FIGURA **5**).

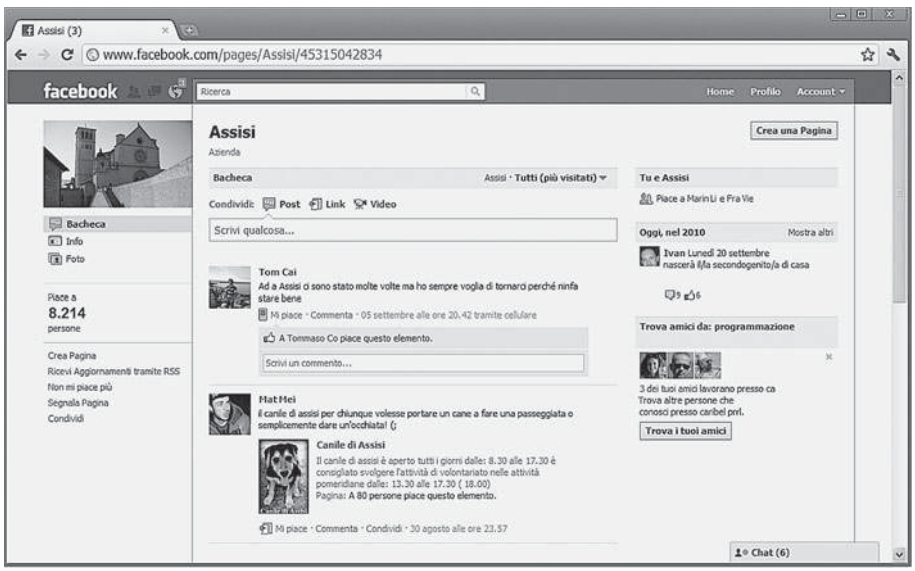


FIGURA **5**

Facebook permette di aggiungere alla propria rete di contatti amici e conoscenti. Ciascuno ha una propria bacheca dove può pubblicare link interessanti, notizie e pensieri, fotografie o note. Ognuna di queste attività comporta la creazione di un contenuto e, se si fa attenzione, la sua aggiunta avviene senza ricaricare la pagina e, almeno questo è l'effetto visuale, «inserendo» il contenuto in cima alla propria bacheca e facendo spostare il resto verso il basso. Non solo: c'è un'ulteriore pagina dove vengono mostrati tutti i contenuti aggiunti dagli utenti che appartengono alla nostra rete di amicizie. Tale pagina viene anch'essa aggiornata **dinamicamente**; infatti,

Facebook

Nelle università statunitensi i «facebook» sono album (stampati o virtuali) con le fotografie degli studenti della scuola. L'ideatore di Facebook, come applicazione web, ha tratto spunto da essa per creare un'applicazione vincente sia in termini di diffusione, che di permanenza sulle pagine.

Quest'ultimo aspetto, insieme al fatto che è possibile conoscere alcuni dati anagrafici degli utenti quali sesso, età, interessi e così via, permette di proporre contenuti pubblicitari mirati e che con alta probabilità vengono letti.

Ecco che da un'applicazione del tutto gratuita l'ideatore guadagna grazie alla pubblicità. Alcune aziende hanno iniziato a guardare con sospetto a tale applicazione e a impedire l'accesso a Facebook ai propri dipendenti nell'orario di lavoro, in quanto causa di distrazione, perdita di tempo e, in definitiva, calo della produttività. Bisogna comunque ammettere che il fenomeno ha cambiato alcune regole sociali e il modo di fare marketing sul web.

mentre si è connessi all'applicazione e qualche nostro amico aggiunge un contenuto sulla sua bacheca, esso viene mostrato in tempo reale in cima alla pagina.
A questo punto è necessario riflettere su quali siano i vantaggi dell'evitare di ricaricare l'intera pagina e perché quasi tutte le moderne applicazioni funzionino in questo modo.
Il motivo principale è che ricaricare un'intera pagina significa farla «scomparire», anche solo per poco, per poi crearla di nuovo. Questo ha un effetto negativo su chi guarda la pagina. Inoltre una pagina molto complessa, fatta per esempio di tante immagini, o di altri oggetti multimediali, necessita di un certo *tempo* per essere caricata e mostrata. Viceversa, l'aggiunta di una piccola novità a una pagina esistente avviene quasi senza che l'utente noti alcun rallentamento né attesa.
Esistono anche altre funzionalità di Facebook realizzate con JavaScript (messaggi tipo email, la possibilità di commentare i contenuti degli amici e così via) ma la caratteristica più evidente è la *chat*. Una volta instaurata una comunicazione con questo strumento, l'invio dei messaggi propri e la ricezione di quelli scritti dall'amico avvengono in *tempo reale*. Questo è possibile grazie a una vera e propria applicazione JavaScript che comunica con il server quello che l'utente digita e riceve eventuali aggiornamenti delle persone coinvolte.

OSSERVAZIONE Spesso, parlando di **Wikipedia** o, in generale, di altre fonti di informazione presenti sul web, capita di chiedersi quale sia la qualità delle informazioni inserite. Per siti così ampi e dalla partecipazione di community così estese, la risposta non è semplice né scontata. Come consiglio generale è bene controllare sempre la veridicità di una notizia o di un'informazione, verificandola anche con altre fonti. Questo è un problema ben noto per tutti i progetti collaborativi; la verifica dei contenuti è demandata ai membri della comunità, ma a volte può non essere del tutto soddisfacente.
L'uso stesso delle ricerche su Google deve essere fatto in maniera razionale, chiedendosi sempre se la fonte (sito web) sia affidabile o meno.
Chiaramente questo vale per qualsiasi fonte informativa; l'autorevole rivista *Nature* ha condotto un'interessante studio comparativo tra alcune voci riportate nell'*Encyclopaedia Britannica* e su Wikipedia; i risultati di tale indagine, per quanto parziali e «a campione», ponevano comunque in buona luce il progetto Wikipedia ma, soprattutto, mostravano come qualsiasi fonte informativa potesse contenere inesattezze e refusi.

2.2 Applicazioni JavaScript che simulano un desktop

Grazie a JavaScript è possibile realizzare applicazioni che simulano un desktop con tanto di icone, finestre che si aprono e si possono trascinare e spostare e così via, tutto dentro una pagina web. Applicativi di questo ge-

Accessibilità e risorse in rete

Alcuni siti seguono una filosofia di progetto che li vuole graficamente molto scarni e poco accattivanti ma, nella loro semplicità, utilizzabili da qualunque browser o dispositivo di lettura delle pagine web.

Questi siti si concentrano principalmente sull'aspetto informativo piuttosto che su quello esteriore.

Uno dei maggiori esperti sul tema di accessibilità del web, Jakob Nielsen, ha formulato critiche spesso severe nei confronti di siti popolari argomentando il fatto che questi puntino troppo su caratteristiche quali le animazioni e la grafica a spese della loro usabilità, specie da parte dei disabili.

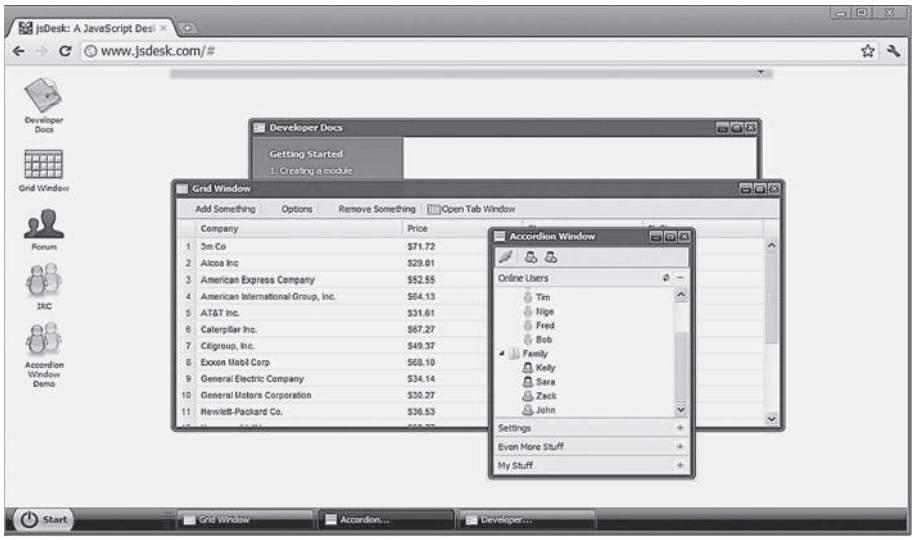


FIGURA 6

nere presentano interfacce con caratteristiche simili a quella mostrata in FIGURA 6.

Con la stessa metafora del desktop si stanno moltiplicando anche strumenti per la collaborazione a distanza. In particolare quelli che permettono l'erogazione e la fruizione di corsi di formazione via web (*e-learning*), grazie anche all'uso di strumenti JavaScript lato client (FIGURA 7).

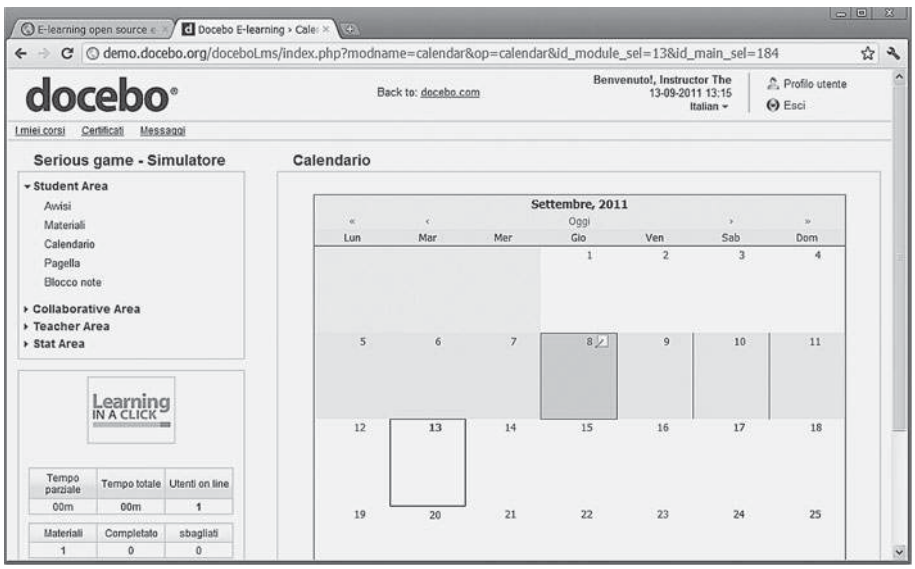


FIGURA 7

2.3 Non solo JavaScript

Realizzare applicazioni sofisticate fruibili sul web è stata sempre un'esigenza molto sentita e, per casi particolari in cui si desiderava una forte interazione, la soluzione di usare HTML e JavaScript non era ritenuta particolarmente efficace, sia per i già citati problemi di compatibilità, sia per la relativa immaturità delle prime versioni del linguaggio JavaScript.

Pertanto diverse aziende hanno realizzato programmi che venivano installati sul browser, attraverso il meccanismo dei *plug-in*, e permettevano l'esecuzione di applicazioni scritte appositamente per tali ambienti. Uno dei più famosi è **Flash**, ma si possono citare anche le *Applet* Java, fino alla recente tecnologia di casa Microsoft chiamata *SilverLight*.
Tutto questo serve per comprendere come il web sia ancora oggi un universo disseminato da tante tecnologie e soluzioni alternative, nessuna davvero dominante sull'altra.

OSSERVAZIONE AJAX, acronimo per *Asynchronous JavaScript And XML*, insieme alla versione 5 dell'HTML, ha momentaneamente decretato il successo dell'accoppiata HTML + JavaScript sulle altre tecnologie e, in particolare, sui diversi plug-in. Vedremo nel prossimo futuro se tale situazione persisterà, magari con nuove aggiunte a JavaScript, o se nasceranno altri plug-in innovativi in grado di contrastare quello che, per ora, sembra uno standard consolidato. Esistono ancora situazioni di nicchia dove si preferisce l'uso di plug-in. Tra i più diffusi c'è senz'altro Flash.

Per inciso anche Flash, la cui esecuzione richiede un plug in, possiede un proprio linguaggio di scripting chiamato *ActionScript*. Al di là del nome, esso è una variante del JavaScript. Non solo: Adobe ha adottato lo stesso linguaggio di scripting anche per altri suoi prodotti. Così è possibile, per esempio, includere codice JavaScript in documenti PDF.

2.4 Lavorare «non in linea»

L'HTML 5 permette di lavorare anche *non in linea* (*offline*). Questo significa che si può accedere in rete a un'applicazione, composta da certe risorse (HTML, CSS e JavaScript), e poi la si può utilizzare anche senza avere una connessione attiva. A questo punto opportuni flag ed eventi vengono innescati ed è compito di chi ha realizzato l'applicazione mantenere uno stato locale e sincronizzarlo con le risorse in rete non appena la connessione viene ripristinata. Questa è una delle scommesse del prossimo futuro, anche per evitare che tutta una serie di dispositivi palmari (smartphone, net book o altro) debbano essere sempre connessi per utilizzare applicazioni web.

2.5 Se non lo puoi battere, fattelo amico

Adobe ha proposto un ambiente di esecuzione da installare sul proprio PC chiamato **Adobe AIR** (*Adobe Integrated Runtime*) che cerca di coniugare il mondo web e le sue tecnologie (HTML e JavaScript *in primis*) con Flash e Flex, altri due strumenti di Adobe. Si tratta essenzialmente di un ambiente di esecuzione locale diverso da un browser. Le applicazioni devono essere scaricate e installate e poi eseguite completamente in locale. Il vantaggio è che chi sviluppa applicazioni riusa tecnologie web standard ma ha a dispo-

JavaScript «fuori-legge»?

In molti stati (primi tra tutti gli USA) sono previste leggi per far sì che portatori di disabilità o altri specifici problemi (daltonici, ipovedenti, …) possano comunque fruire delle informazioni pubblicate sui siti web. Strumenti di navigazione meno sofisticati degli usuali browser che magari interpretino i siti come «solo testo», senza formattazione, script o plug-in possono essere la risposta ad alcuni dei problemi citati.
In Italia la cosiddetta «Legge Stanca» (dal nome del suo promotore) obbliga gli enti pubblici a rispettare regole di accessibilità dei propri siti e auspica che queste siano recepite anche da aziende e privati.
Uno dei punti di tale legge riguarda proprio l'uso parsimonioso degli script. Questi dovrebbero «arricchire» e migliorare la navigazione del sito, senza però condizionarne le funzionalità nel caso in cui vengano disabilitati.
In questo senso, nella realizzazione di siti web, è importante studiare attentamente l'utilizzo degli script e delle eventuali alternative possibili.

sizione *strumenti grafici avanzati* e un controllo molto maggiore rispetto a quello fornito da un browser per le risorse locali.

2.6 JavaScript e la sicurezza

JavaScript non ha alcun *accesso alle risorse locali*, siano esse dischi di memoria esterni o dati dei programmi in memoria principale. Questo evita che possibili virus o altri programmi malevoli possano carpire informazioni riservate. In ogni caso tutti i browser permettono, volendo, di disabilitare l'esecuzione degli script.

OSSERVAZIONE Verificare se un'applicazione presenta dei rischi di sicurezza è un compito da specialisti. Google, nel 2011, ha creato un'estensione per Chrome che permette di analizzare in automatico eventuali problemi di sicurezza nell'ambito del progetto *DOM Snitch*.

3 Un *excursus* storico e i fondamenti del linguaggio

Dall'introduzione risulta evidente che quello che oggi viene comunemente chiamato JavaScript ha avuto una evoluzione piuttosto travagliata. Vediamo brevemente la sua storia.

Netscape stava cercando di creare un nuovo linguaggio di programmazione adatto sia a essere usato dentro le pagine HTML, sia come possibile linguaggio per applicazioni server. Brendan Eich fu, nel 1995, lo sviluppatore incaricato della creazione di questo linguaggio. In origine il nome del nuovo linguaggio doveva essere *LiveScript*, dato dall'unione di due parole, *Live* e *Script*, che in qualche modo ricordavano la sua funzione (*Live*, nel senso di «dare vita» alle pagine HTML altrimenti statiche) e al modo di usarlo (*Script*, ovvero all'interno di una pagina HTML e non come programma autonomo). Ma a quel tempo si stava affacciando sul mercato un nuovo linguaggio, Java, della Sun Microsystems (ora acquisita da Oracle), che prometteva meraviglie nella creazione di applicazioni (applet) attraverso un plug-in da installare sui browser.

La scelta di privilegiare una tecnologia piuttosto che un'altra ha trovato una contrapposizione netta tra i sostenitori delle diverse opzioni. Il risultato è stato uno strano compromesso: mantenere entrambe le tecnologie e abbandonare il nome LiveScript in favore di **JavaScript** per il nuovo linguaggio di scripting; ancora oggi, questa scelta porta a molta confusione a chi si avvicina alla programmazione web e pensa che Java e JavaScript siano la stessa cosa. In realtà Java oramai ha trovato il suo ambito di applicazione più per applicazioni server o desktop piuttosto che per la scrittura di applet, mentre JavaScript ha avuto nuovo vigore, per la sua semplicità ma anche per la sua duttilità, ed è un linguaggio in piena evoluzione e ampiamente utilizzato in tutte le applicazioni web.

Come nasce un linguaggio

La nascita di un nuovo linguaggio dovrebbe essere preceduta da studi approfonditi e verifiche di utilizzo prima di un rilascio ufficiale. Talvolta, però, le logiche di mercato hanno il sopravvento sul buon senso e sulle regole di corretta gestione di un progetto.

JavaScript, in particolare, nasce da uno studio preliminare e la creazione di un prototipo fu fatta in circa una settimana. Anche i tempi del suo rilascio sono stati particolarmente stretti e dettati più dal desiderio di «avere un linguaggio sul browser» che dall'attenzione alla solidità delle sue basi.

All'inizio JavaScript veniva usato sia per migliorare l'aspetto estetico delle pagine, sia per aiutare gli utenti a ottimizzare la fruizione delle pagine. Per esempio, fu da subito sfruttata la capacità di JavaScript di interagire con il contenuto delle pagine per creare effetti *rollover*: con questo termine si intende il cambiamento di un'immagine quando il cursore del mouse si posiziona sopra un'immagine esistente. Ecco pertanto che ci sono menu le cui voci sono costituite da immagini, per esempio con un font monocolore; le voci vengono colorate quando il cursore è sopra di esse, oppure si attivano altri effetti grafici.

ESEMPIO

Per verificare subito come ciò sia possibile basta fare la seguente prova: aprire Paint o un altro programma di disegno, e creare un'immagine con un testo qualsiasi. Salvarla con un'estensione appropriata per il web (vanno bene gif, jpeg o png); per esempio con il nome *prova1.gif*. Dopodiché aggiungere un effetto, per esempio una sottolineatura alla scritta, e salvarla come seconda immagine (*prova2.gif*). Non resta che creare una pagina con estensione .htm (o .html) e digitare il codice che segue:

```
<img src="prova1.gif"
  onmouseover="this.src='prova2.gif'"
  onmouseout="this.src='prova1.gif'" />
```

OSSERVAZIONE L'esempio è una porzione di una pagina HTML. Per verificarlo è necessario avere a disposizione un browser in grado di interpretare l'HTML e il JavaScript eventualmente contenuto. Quest'ultimo requisito non è un problema per gli usuali browser per PC; può esserlo per browser eseguiti da smartphone o palmari, anche se oramai la quasi totalità dei browser interpreta correttamente il codice JavaScript.

Quando il cursore del mouse sarà sopra l'immagine apparirà la scritta sottolineata (o con l'effetto scelto applicato) e quando sarà su un'altra zona l'immagine ritornerà a essere senza alcun effetto grafico.
Nel tempo gli sviluppatori si sbizzarrirono estendendo questa funzionalità di base, per esempio facendo sì che, oltre a cambiare la voce selezionata, anche il testo di un'altra parte della pagina cambiasse: ecco così realizzati semplici aiuti contestuali.

ESEMPIO

```
<img src="prova1.gif"
  onmouseover="document.getElementById('txt').innerHTML='spiegazione primo elemento'" />
<br/>
<img src="prova1.gif"
  onmouseover="document.getElementById('txt').innerHTML='spiegazione secondo elemento'" />
<br/>
<img src="prova1.gif"
  onmouseover="document.getElementById('txt').innerHTML='spiegazione terzo elemento'" />
<br/>
<div id="txt"></div>
```

Altro ambito di utilizzo fu quello di eseguire controlli su quanto l'utente ha digitato nelle form, in modo da controllare che tutti i valori obbligatori fossero inseriti e, possibilmente, fossero corretti.

ESEMPIO

Dopo aver creato un pagina con il JavaScript seguente:

```
<form>
Nome e cognome: <input type="text" id="nome" name="nome" /> <br/>
Data di nascita: <input type="text" name="data" /> <br/>
<input type="button" value="Salva"
onclick="if(document.getElementById('nome').value=='')
alert('Nome e cognome sono obbligatori')
else
alert('Ok, invio i dati')" />
</form>
```

Si provi a premere su «salva» prima senza digitare nulla su nome e cognome; poi si ripeta l'esperimento inserendo qualche valore.

Controllare subito la *correttezza* dei dati aiuta a minimizzare le interazioni con il server, e quindi a ridurre i dati scambiati con esso. Si ricordi che erano anni in cui le connessioni erano molto spesso lente e poco performanti; «guadagni» di questo tipo permettevano all'utente di risparmiare tempo (e denaro) ed erano fortemente consigliate per tutte le applicazioni. La diffusione di abbonamenti flat di oggi e una presenza pressoché capillare di connessioni a banda larga hanno minimizzato l'importanza di dare il prima possibile il feedback all'utente su quanto inserito.

OSSERVAZIONE Si presti attenzione al fatto che validare i lati sul client non esime il programmatore dal rivalidarli sul server. Questa mancanza potrebbe portare, infatti, a rischi di sicurezza sull'applicazione server.

3.1 ECMAScript e la maturità

Fino ad alcuni anni fa era comune l'uso di tabelle per la formattazione delle pagine e a volte venivano scritti frammenti di codice HTML non valido, per esempio perché mancavano i tag principali della struttura, come html, head, body (gli esempi sopracitati sono, volutamente, «pessimi» proprio perché mancanti di tale struttura).
Presto si passò alla creazione di standard del linguaggio html molto più restrittivi (è il caso dell'XHTML, in cui il file, oltre a dover essere un documento XML valido, deve essere anche conforme al foglio di stile della specifica) e di tecnologie che si occupano della sola presentazione dei dati (CSS).
In tale contesto anche il linguaggio JavaScript subì notevoli evoluzioni.
La prima, sebbene più «formale» che sostanziale, è quella di diventare uno standard proposto da un organismo internazionale e le cui regole sono

dettate da una specifica formale. Questa novità tracciò la via maestra per spingere i produttori a mantenere la loro implementazione del linguaggio coerente con le specifiche date.
In secondo luogo il linguaggio stesso adottò nuovi costrutti che lo rendono maggiormente adatto a un contesto dinamico come quello del web. È il caso degli oggetti che permettono di realizzare applicazioni di tipo AJAX o delle API per la manipolazione di documenti XML. Inoltre il forte uso dei CSS ha fatto sì che venisse data sempre più importanza alla possibilità di modificare, oltre alla pagina HTML, anche i CSS e i modi di applicarli ai diversi elementi della pagina direttamente da codice JavaScript.

4 Dentro il linguaggio JavaScript

Una caratteristica molto importante di JavaScript è quella di avere accesso ai contenuti della pagina, sia in lettura sia in scrittura, come pure di poter associare codice al verificarsi di certi eventi.

ESEMPIO

Ecco come generare una pagina HTML che ha, come contenuto, il valore dell'orologio di sistema dove viene eseguito il browser (FIGURA **8**).

```
<html>
<body>
<script type="text/javascript">
  document.write("Ora del sistema " + new Date());
</script>
</body>
</html>
```

FIGURA **8**

ESEMPIO

Ecco come mostrare un messaggio di saluto, che compare alla chiusura della finestra:

```
<html>
<body onclose="alert('Arrivederci!')">
</body>
</html>
```

Un po' di buone maniere

Gli esempi delle pagine precedenti e quello in questa pagina sono stati volutamente ridotti all'osso, scrivendo codice non valido secondo le specifiche HTML. Per evitare questo si consiglia sempre di validare la pagina e imparare dagli errori qual è il modo corretto di scrivere le pagine.
Altre volte ci sono delle finezze stilistiche o la possibilità di specificare o meno attributi opazionali. Per esempio è corretto usare l'attributo type per il tag script (inserendo, come valore "text/javascript") e concludere ciascuna istruzione JavaScript con un punto e virgola. Tale carattere specifica che l'istruzione JavaScript è terminata; se si ha l'accortezza di inserire un'istruzione per linea il suo uso è opzionale; ma anche in questo caso è buona norma ricordarsi di inserire comunque tale carattere di terminazione.

Gli esempi fin qui presentati sono, con tutta probabilità, oscuri nel senso che non se ne conosce bene il funzionamento, non conoscendo nulla del linguaggio. Eppure si può osservare che essi, nella loro semplicità, possono essere copiati e incollati in una qualsiasi pagina web ed essere perfettamente funzionanti! Non solo: chi conosce l'inglese può «intuire» il significato delle singole istruzioni. Questo è uno dei vantaggi del linguaggio: ben si presta a un uso «poco consapevole», perché basta fare copia e incolla da siti che offrono questo tipo di **soluzioni «pronte all'uso»**.
È chiaro che così facendo non si è in grado di creare alcuna funzionalità originale e, molto spesso, l'effetto che si vuol ottenere difficilmente si trova già pronto. È giunto pertanto il momento di entrare un po' più in profondità e di imparare i fondamenti del linguaggio, senza illudersi di saper programmare in JavaScript solo perché si sanno copiare e incollare diversi pezzi presi da script diversi. Per evitare questo si analizzeranno nel dettaglio le basi del linguaggio, per poi approfondire gli aspetti più avanzati e l'interazione con i browser.

4.1 Le caratteristiche peculiari

JavaScript è un linguaggio **interpretato**: i *tipi* sono associati ai *dati* e non alle variabili; pertanto una *variabile* può contenere un valore di qualsiasi tipo e assegnare valori di tipo diverso alla stessa variabile.
JavaScript è, come molti altri linguaggi moderni, *orientato agli oggetti*. Come in altri linguaggi è possibile scrivere programmi che non fanno uso di tale caratteristica ma che prevedono uno stile di programmazione imperativo. Nel seguito esamineremo queste caratteristiche approfondendone l'utilizzo e i vantaggi dei vari approcci.
Il codice JavaScript può essere inserito in qualunque parte del documento HTML o, se si realizzano funzioni di utilità generale, su un documento esterno; in quest'ultimo caso il file esterno deve essere referenziato dentro i tag `<head>` del documento e l'attributo `src` va valorizzato con l'URL del file esterno.

OSSERVAZIONE In JavaScript, come in Java, è possibile utilizzare due tipi di commenti: il primo è preceduto dai caratteri // e inizia con essi per terminare al primo «a capo». Il secondo tipo inizia con /* e termina con */; al suo interno quest'ultimo tipo di commenti può contenere qualsiasi carattere, anche «a capo», ma non si possono innestare nuovi commenti dello stesso tipo.
Come sempre è buona norma commentare le parti più significative del codice, per agevolare la sua modifica a distanza di tempo, rendendo più comprensibile il flusso del programma. Meglio evitare commenti per le istruzioni autoesplicative. Per esempio, è del tutto inutile il commento del primo esempio del prossimo paragrafo perché per chi conosce le regole di base di JavaScript esso non aggiunge alcuna informazione all'istruzione commentata; invece è importante commentare le parti il cui scopo non è chiaro a una prima e rapida occhiata.

ESEMPIO

```
<html>
<head>
  <script src="./urlrelativa.js"
    type="text/javascript" language="javascript"></script>
  <script src="http://www.example.org/urlassoluta.js"
    type="text/javascript"></script>
</head>

</html>
```

4.2 Variabili e tipi di dato elementari

Nella TABELLA 1, sulla prima colonna, sono elencati i tipi di dato primitivi e il tipo di dato Object; la seconda colonna specifica quali valori sono ammessi per ciascun tipo.

TABELLA 1

Tipo	Valori
undefined	**undefined**
Null	**null**
Boolean	**true**, **false**
String	Qualsiasi stringa
Number	Qualsiasi numero; in particolare i numeri sono sempre dei numeri decimali a doppia precisione a 64 bit, i cui valori sono definiti dallo standard IEEE 754
Object	Un qualsiasi oggetto

È buona norma *dichiarare* sempre le variabili e assegnare a esse un *valore iniziale*. Se una variabile non è dichiarata, un tentativo di accedere al suo valore in lettura genera un *errore*; se non viene dichiarata ma si accede in scrittura (per esempio assegnando a essa un valore), allora la dichiarazione risulta *implicita* e ha visibilità globale.

ESEMPIO

```
<script type="text/javascript">
  esempio = 0; // crea e inizializza una nuova variabile globale
  alert(test); // errore
</script>
```

Se si dichiara una variabile e non le si assegna alcun valore, essa contiene un valore particolare che è ***undefined***. La dichiarazione avviene specificando la parola chiave *var* seguita dal nome della variabile e, volendo, l'assegnamento al suo valore iniziale.

ESEMPIO

```
<script type="text/javascript">
  var nome = 0; // dichiarazione e inizializz. della variabile
                // 'nome'
  var altra;
  alert(altra); // mostra undefined
</script>
```

Una variabile non ha visibilità globale quando viene dichiarata dentro una funzione usando la parola chiave var. In quest'ultimo caso essa ha visibilità solo dentro la funzione che la dichiara.
Una funzione viene indicata con la parola chiave function, seguita dal nome della funzione, con i parametri della funzione, dichiarati all'interno di parentesi tonde, e, a seguire, le parentesi graffe che ne racchiudono il corpo.

ESEMPIO

```
<script type="text/javascript">
  var globale = 0;
function funzione(){
       var locale = 0; // qui inizia la visibilità per 'locale'
       /*
       altre istruzioni
       */
     esempio = 0; //crea e inizializza una nuova variabile globale
} // qui termina la visibilità per 'locale'
</script>
```

Il valore *null* indica l'assenza di un qualsiasi valore che, altrimenti, sarebbe un oggetto.
Gli oggetti possono essere di tre tipi: **nativi** (o **built-in**), **forniti dall'host** e **definiti dagli utenti**.
Quelli nativi sono completamente definiti dallo standard, indipendentemente dall'ambiente di esecuzione; un esempio è l'oggetto `Date` dell'esempio visto in precedenza della pagina HTML con l'orologio di sistema.
Quelli forniti dall'host dipendono da dove il programma JavaScript viene eseguito: se dentro un browser web, l'host fornirà oggetti che permettono di interagire con la pagina e il browser stesso; se dentro un PDF, l'host fornirà oggetti che espongono le funzionalità e le proprietà del documento e così via.
Infine, quelli definiti dagli utenti sono oggetti che ciascun programmatore costruisce per meglio modellare la soluzione al problema che ha deciso di risolvere.

4.3 Lavorare con i numeri

In JavaScript è possibile rappresentare **numeri interi** e **reali** ma, al di là del modo di scriverli, sono sempre rappresentati come numeri reali a **doppia precisione**. I numeri interi sono esprimibili come sequenze di cifre, i cui valori vanno da -2^{53} a 2^{53}. Un numero può anche essere rappresentato in

base 16; per farlo esso inizia per 0x o 0X; pertanto i numeri che seguono sono equivalenti (l'operatore === verrà introdotto tra breve; basti sapere che è un operatore che verifica l'uguaglianza degli argomenti).

ESEMPIO

```
<script type="text/javascript">
  var a = 100;
  var b = 0x64;
  var c = 0X64;
  alert(a===b && b===c); // mostra true
</script>
```

Un numero *reale* può essere rappresentato indifferentemente sia esprimendolo come parte intera, un punto e a seguire le cifre decimali, sia usando la notazione esponenziale.

ESEMPIO

Ecco un modo per esprimere lo stesso numero reale:

```
<script type="text/javascript">
  var a = 0.00234;
  var b = 2.34e-3;
  alert(a===b); // mostra true
</script>
```

Esistono valori particolari per i numeri: `Infinity` indica l'infinito positivo, mentre la notazione `-Infinity` indica l'infinito negativo, `NaN` (*Not-a-Number*) è un valore di tipo numerico che indica un numero non valido.

ESEMPIO

```
<script type="text/javascript">
  var infinito = 5 / 0;     // vale Infinity
  var nonnumero = 3 / 'a'; // vale NaN
</script>
```

Nella TABELLA **2** sono indicati gli operatori matematici del JavaScript.

TABELLA **2**

Operatore	Descrizione
*	Moltiplicazione
/	Divisione
%	Modulo
+	Somma
-	Sottrazione (esiste sia la forma binaria sia la forma unaria)

Come di solito, gli operatori di moltiplicazione, divisione e modulo hanno la precedenza sugli altri. Unica eccezione la forma unaria della sottrazione (cambio di segno).

Apici e doppi apici

JavaScript delimita le stringhe o con i doppi apici o con gli apici singoli. La scelta è del tutto equivalente.

L'HTML, invece, utilizza sempre i doppi apici per delimitare i valori degli attributi dei suoi tag; per questo motivo è comune utilizzare gli apici singoli nel caso in cui il codice JavaScript venga inserito dentro un attributo HTML.

Uno degli errori più diffusi nelle pagine web è quello di utilizzare le *HTML entities* anche dentro le stringhe JavaScript. Usare, per esempio, *è* per rappresentare una «e» con l'accento grave è corretto nel caso di codice HTML, mentre è sbagliato come parte di una stringa JavaScript. In quest'ultimo caso o si usa il carattere con la lettera accentata o si inserisce il codice Unicode corrispondente.

Quando si devono inserire degli apici singoli o doppi dentro una stringa, bisogna verificare sempre se questi sono uguali al carattere usato per delimitare la stringa; se è così il carattere contenuto va usato con una backslash davanti (si dice che viene fatto l'escape del carattere).

ESEMPIO

```
<script type="text/javascript">
  var espr = 2 * 10 % 3 - 7 / 2;  // vale -1.5
  var equiv= (2 *(10 %3))-(7/2);  // vale -1.5
</script>
```

Anche in JavaScript esistono i classici operatori unari di incremento e decremento: ++ e --. Essi possono precedere una variabile o seguirla. Nel primo caso prima viene fatto l'incremento, poi viene restituito il valore se fa parte di un'espressione. Nel secondo caso, prima restituiscono il valore e poi fanno l'incremento.

ESEMPIO

```
<script type="text/javascript">
  var x = 10;
  var y = x++;
  alert(x);   // mostra 11
  alert(y);   // mostra 10
  x = ++y;
  alert(x);   // mostra 11
  alert(y);   // mostra 11
</script>
```

4.4 Lavorare con stringhe di caratteri

I **caratteri** JavaScript sono rappresentati secondo la codifica **Unicode**. Una stringa di caratteri è un insieme di caratteri racchiusi tra apici singoli o doppi.

ESEMPIO

```
<script type="text/javascript">
  var str1 = "una stringa";
  var str2 = 'altra stringa';
</script>
```

Il carattere di backslash (\) è utilizzato per rappresentare caratteri speciali; l'esempio che segue indica come rappresentare un «a capo» alla fine di una stringa.

ESEMPIO

```
<script type="text/javascript">
  var str = "Questa è una riga\n";
</script>
```

Nella TABELLA **3** è riportata una sintesi dei principali caratteri speciali previsti e il modo per rappresentare un carattere Unicode conoscendo il suo codice.

TABELLA 3

Carattere	Significato
\\	Carattere di backslash
\'	Apice singolo (utile dentro stringhe delimitate da apici singoli)
\"	Apice doppio (utile dentro stringhe delimitate da apici doppi)
\b	Backspace
\f	Form feed
\n	Nuova riga
\r	Ritorno carrello
\t	Tabulazione orizzontale
\v	Tabulazione verticale
\uNNNN	Carattere Unicode il cui codice è il numero NNNN

Una delle operazioni più comuni è la concatenazione di stringhe, che viene realizzata usando l'operatore infisso +:

ESEMPIO

```
<script type="text/javascript">
  alert('Un'+' '+'esempio');
</script>
```

4.5 Espressioni e variabili

Un'**espressione** è la rappresentazione di un calcolo che restituisce un valore. Essa può essere composta da *costanti* (numeriche, stringhe o valori di verità) e *operatori*. Una **variabile** può memorizzare un valore qualunque e permette di referenziare lo stesso dentro altre espressioni o parti di programma.

ESEMPIO

```
<script type="text/javascript">
  var variabile = 5 + 3*2 - 4;
  alert(variabile+1);
</script>
```

OSSERVAZIONE In generale, per *valutare* un'espressione si deve tener conto della *precedenza* degli operatori; a parità di precedenza si valuta quello più a sinistra. Eventuali *parentesi tonde* permettono di variare l'ordine di valutazione secondo le regole usuali: le espressioni delimitate dalle parentesi più interne vengono valutate prima. Se si hanno dubbi sulle regole di precedenza applicate dall'interprete, è meglio mettere le parentesi anche quando non sono necessarie.

4.6 Modifiche al flusso e tipi di dato booleani

Si è già avuto modo di osservare come, in un qualunque insieme, le istruzioni JavaScript vengano eseguite una dopo l'altra; per eseguire alcune determinate istruzioni solo in presenza di precise condizioni, si può ricorrere al costrutto *if/then/else*.

ESEMPIO

```
<script type="text/javascript">
  var condizione = true;
  if (condizione)
  {
    /* porzione di codice eseguita
    quando la condizione è vera
    */
  } else {
    /* porzione di codice eseguita
    quando la condizione è falsa
    */
  }
</script>
```

In Javascript una *condizione* può essere una qualunque *espressione* di *tipo booleano*. Tale tipo può assumere due valori, identificati in JavaScript dalle costanti **`true`** e **`false`**, che rappresentano, rispettivamente, i valori di verità «vero» e «falso».
Anche qui le parentesi che racchiudono la condizione sono obbligatorie e il ramo *else* è opzionale e può essere omesso. Dopo il ramo *else* ci può essere ancora un'istruzione *if*: le istruzioni rimanenti vengono eseguite solo se valgono altre specifiche condizioni.

ESEMPIO

```
<script type="text/javascript">
  var condizione = ..., altra = ...;
  // condizione e altra = true o false
  if (condizione)
  {
    /* porzione di codice eseguita
    quando condizione è vera
    */
  } else if (altra) {
    /* porzione di codice eseguita
    quando condizione è falsa e altra è vera
    */
  }
</script>
```

Gli operatori ammessi per le espressioni booleane sono di quattro tipi: operatori booleani, operatori di uguaglianza, operatori di identità e operatori di relazione d'ordine.

4.7 Operatori booleani

Gli usuali operatori logici sono rappresentati come in TABELLA 4 (la precedenza degli operatori è la stessa del loro ordine di presentazione; per modificare l'ordine di valutazione si possono usare le usuali parentesi tonde).

TABELLA 4

Operatore	Significato
!	Negazione logica, cambia il valore da `true` a `false` e viceversa
&&	AND logico: vale `true` solo quando entrambi gli operandi lo sono
||	OR logico: vale `true` quando almeno uno degli operandi lo è

È importante osservare che quando viene valutata un'espressione che fa uso di operatori booleani, l'interprete interrompe la valutazione dell'espressione non appena è in grado di conoscere il risultato.

ESEMPIO

Dato che l'operatore || restituisce `true` se almeno uno degli operandi lo è, valuta il primo operando e se esso è `true` restituisce il risultato senza valutare il secondo; questo è evidente dal seguente programma:

```
<script type="text/javascript">
  var x = 10;
  var y = 10;
  alert (++x > y || ++y > x); // mostra true
  alert(x); // mostra 11
  alert(y); // mostra 10
</script>
```

In modo analogo l'operatore && restituisce `false` se almeno uno degli operandi lo è; se è `false` il primo, non valuta il secondo.

4.8 Operatori di uguaglianza

Per verificare se due valori sono uguali si usa l'operatore di uguaglianza denotato con ==. Esso si usa con notazione infissa e accetta due valori (o espressioni) di qualsiasi tipo. Restituisce `true` se gli operandi hanno lo stesso valore.

ESEMPIO

```
<script type="text/javascript">
  var x = 10;
  var y = 10;
  alert(x==y); // mostra true
</script>
```

L'operatore == restituisce `true` anche se i tipi di dato degli operatori sono diversi, ma vengono convertiti nello stesso valore.
Infatti il test di uguaglianza tra due numeri o espressioni numeriche restituisce `true` solo se i due valori sono gli stessi. Se si confrontano due stringhe, esse sono uguali solo se composte dagli stessi caratteri (dove le lettere maiuscole e minuscole non sono fra loro equivalenti!). Il confronto tra due espressioni booleane restituisce `true` solo se gli argomenti vengono valutati con lo stesso valore di verità. Ma se si confrontano tipi di dato diversi, JavaScript tenta delle *conversioni* (o *cast*) sui valori prima di dare il risultato.

ESEMPIO

```
<script type="text/javascript">
  var x = 10;
  var y = "10";
  alert(x==y); // mostra true
</script>
```

Le conversioni fra tipi verranno analizzate più avanti.
Accanto all'operatore di uguaglianza c'è quello di non uguaglianza, che si esprime con `!=`. Esso è equivalente alla negazione del risultato del test di uguaglianza: `a != b` equivale a `!(a==b)`.

4.9 Operatori di identità

Se si vuol verificare che due valori siano gli stessi al di là di possibili conversioni, si ricorre all'operatore di identità espresso con ===. Quando due valori o espressioni sono di tipo diverso, automaticamente l'operatore === restituisce `false`.

ESEMPIO

```
<script type="text/javascript">
  var x = 10;
  var y = "10";
  alert(x===y); // mostra false
</script>
```

4.10 Operatori di relazione d'ordine

Gli usuali operatori di relazione d'ordine sono sintetizzati nella TABELLA 5.

TABELLA 5

Operatore	Descrizione
<	Minore (stretto)
<=	Minore o uguale
>	Maggiore (stretto)
>=	Maggiore o uguale

OSSERVAZIONE La *relazione d'ordine* dei numeri è quella usuale; per i caratteri vale l'ordine *lessicografico* con i caratteri maiuscoli che precedono quelli minuscoli. Per i valori di verità booleani si assume che il valore `true` sia maggiore di `false`.

ESEMPIO

```
<script type="text/javascript">
  alert(1<8);            // mostra true
  alert('a' <= 'z');     // mostra true
  alert (true > false);  // mostra true
</script>
```

OSSERVAZIONE Si faccia attenzione che gli operatori di relazione d'ordine di JavaScript sono realizzati unicamente per il tipo di dato numerico e stringa. Se viene usato un operando di tipo diverso, esso viene *convertito* in uno dei due tipi per poter decidere il risultato.

4.11 Conversioni implicite ed esplicite tra tipi

Come si è avuto modo di osservare JavaScript *non è fortemente tipizzato* e in molte occasioni applica **conversioni implicite**. In particolare valgono le seguenti regole.

- Se in un determinato contesto viene richiesta una stringa e viene usato un numero, allora esso viene trasformato nella rappresentazione in stringa del numero stesso. Se si usa `true` o `false` essi vengono trasformati, rispettivamente, nelle stringhe `'true'` o `'false'`.

ESEMPIO

```
<script type="text/javascript">
  alert('a' + 12 + 1); // mostra 'a121'
  alert('a' + true);   // mostra 'atrue'
</script>
```

- Se viene richiesto un numero e viene usata una stringa, allora JavaScript tenta di usare il numero contenuto nella stringa; se la stringa è la stringa vuota, essa viene trasformata nel numero 0; se viene usato un valore di verità, esso viene trasformato in 0 se `false`, in 1 altrimenti.

ESEMPIO

```
<script type="text/javascript">
  alert('12'*2);  // mostra 24
  alert('12'+2);  // mostra '122' per la regola precedente
  alert(1+true);  // mostra 2
</script>
```

- In un contesto dove viene richiesto un valore booleano, vengono valutati a `true` qualunque stringa non vuota e qualunque numero diverso da 0; se la stringa è vuota o il numero vale 0, viene trasformato a `false`;

ESEMPIO

```
<script type="text/javascript">
  if(11)
    alert('ok') // mostra ok
  else
    alert('ko');
</script>
```

Nel caso si vogliano applicare delle conversioni in maniera esplicita, si può ricorrere a *Number* per la conversione a numero, *String* per la conversione a stringa e *Boolean* per la conversione a valore di verità.

ESEMPIO

```
<script type="text/javascript">
  Number("21"); // 21
  String(29);   // "29"
  Boolean(2);   // true
</script>
```

4.12 Funzioni

In generale con le parentesi graffe si racchiudono blocchi di programma. Ma particolari blocchi possono essere definiti per essere richiamati da qualsiasi parte del programma: sono le **funzioni**.

ESEMPIO

```
<script>
  function mostraOrario(){
    alert(new Date());
  }
  mostraOrario();
</script>
```

Una funzione può ricevere valori al momento dell'invocazione tramite i suoi parametri.

ESEMPIO

```
<script>
  function mostraOrario(scritta){
    alert(scritta + ' ' + new Date());
  }
  mostraOrario('Oggi è');
</script>
```

Una funzione può restituire un valore, da usare al posto di variabili, valori o espressioni.

ESEMPIO

```
<script>
  function area(raggio){
    return raggio*raggio*3.14;
  }
  var raggio = 3;
  alert('area cerchio='+area(raggio));
</script>
```

Se si invoca una funzione con meno argomenti dei suoi parametri, l'invocazione va a buon fine, ma ai parametri non specificati viene assegnato il valore *undefined*.

ESEMPIO

```
<script>
  function test(a, b, c){
    return a + ' ' + b + ' ' + c;
  }
  alert(test(1, 2, 'A'));  // mostra "1 2 A"
  alert(test(1));          // mostra "1 undefined undefined"
</script>
```

5 Vettori, iterazioni e cicli

Finora abbiamo analizzato tipi di dato scalari; ora si vedrà come dichiarare e utilizzare tipi di dato vettoriali.

5.1 Vettori

Un vettore, o array, è, anche qui, un **insieme ordinato di elementi**. JavaScript permette di definire un vettore specificando i valori che può assumere tra *parentesi quadre*.

ESEMPIO

```
<script type="text/javascript">
  var vuoto = [];
  var vett = ['q', 1, true];
</script>
```

Per conoscere la dimensione di un vettore è possibile fare riferimento alla solita proprietà *length* del vettore.

ESEMPIO

```
<script type="text/javascript">
  alert(['q', 1, true].length); // mostra 3
</script>
```

OSSERVAZIONE Molti linguaggi di programmazione ammettono solo vettori *omogenei*, ovvero i cui elementi sono di un solo tipo di dato. JavaScript ammette **vettori non omogenei**, senza alcun vincolo sul tipo dei dati contenuti.

Per accedere a un qualsiasi elemento è necessario riferirsi alla sua posizione; pertanto si userà il nome del vettore seguito da un numero, che è la posizione dell'elemento, racchiuso tra parentesi quadre.

ESEMPIO

```
<script type="text/javascript">
  var giorno_settimana=['Lunedì', 'Martedì', 'Mercoledì',
                        'Giovedì', 'Venerdì', 'Sabato',
                        'Domenica'];
  alert(giorno_settimana[3]); // mostra Giovedì
</script>
```

È possibile creare nuovi elementi di un array assegnando un valore ai suoi elementi.

ESEMPIO

```
<script type="text/javascript">
  var v = [];
  v[0] = 1;
  v[1] = 1;
  v[2] = 1;
  v[3] = 1;
  alert(v.length); // mostra 4
</script>
```

È piuttosto comune assegnare un valore di *default* a tutti gli elementi di un vettore; ma al crescere degli elementi è piuttosto scomodo farlo uno per uno. In questo caso è utile un costrutto che permetta di «scorrere» i diversi elementi in maniera da scriverli (o leggerli) uno dopo l'altro.

5.2 Iterazioni e cicli

In JavaScript esistono quattro tipi di ciclo: i classici *while*, *do/while* e *for* e uno che permette di iterare su tutti gli elementi di un insieme (*for each*).

ESEMPIO

Il ciclo *while* assume la tipica forma:

```
while(condizione)
    corpoDelCiclo;
```

Quando si desidera che il ciclo venga eseguito almeno una volta, indipendentemente dal valore iniziale della condizione, è necessario usare un ciclo con condizione in coda, che assume forma classica: *do* / istruzione / *while* (espressione);

ESEMPIO

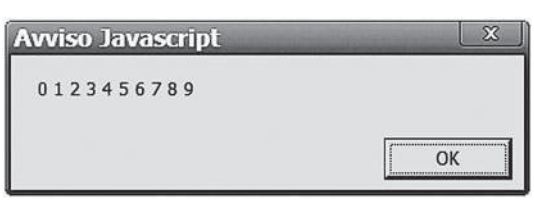
Ecco un programma JavaScript che simula il seguente comportamento di un turista: fa un viaggio e, se gli è piaciuto, ne fa un altro, altrimenti si ferma.

```
<script type="text/javascript">
  var bella_esperienza;
  var numero_viaggi = 0;
  do{
       numero_viaggi++;
       bella_esperienza = prompt("Piaciuto s/n?")=='s'
  } while(bella_esperienza);
  alert('Hai fatto' + numero_viaggi + 'viaggi');
</script>
```

Anche in JavaScript, quasi sempre la condizione si basa sulla verifica del valore di una variabile, comunemente indicata come *variabile di ciclo* o *contatore*. Questa variabile viene inizializzata prima di entrare dentro il ciclo; la condizione del ciclo la utilizzerà per decidere se fare un'altra iterazione oppure no, e alla fine del ciclo, prima di rieseguire una nuova iterazione, la variabile di ciclo verrà aggiornata.

ESEMPIO

```
<script type="text/javascript">
  var mostra = "";
  var i = 0;        // i è la variabile di ciclo, qui inizializzata
  while(i < 10) {   // fa parte della condizione di terminazione
    mostra = mostra + " " + i;
    i++;            // viene aggiornata
  }
alert(mostra);
</script>
```



Alert e prompt

Alert è una funzione che mostra un risultato ed è già stata vista e usata in più punti; *prompt* è la funzione che mostra una finestra con una domanda e permette all'utente di inserire una risposta; tale risposta viene restituita dalla funzione a chi l'ha invocata.

Entrambe le funzioni sono poco usate in programmi reali, perché poco personalizzabili a livello di interfaccia. Molto meglio utilizzare le form HTML e gestirle attraverso JavaScript, se si vuole lasciare all'utente la possibilità di inserire valori.

Anche qui, il costrutto *for* riassume questi tre momenti (inizializzazione della variabile di ciclo, test e suo incremento) in un'unica istruzione.

ESEMPIO

```
<script type="text/javascript">
  var mostra = "";
  for(var i=0; i<10; i++)
    mostra = mostra + " " + i;
  alert(mostra);
</script>
```

Ecco come dichiarare un vettore e inizializzare tutti i valori, facendo sì che il valore dell'elemento sia uguale alla sua posizione all'interno del vettore.

ESEMPIO

```
<script type="text/javascript">
  var v = [];
  for(var i=0; i<10; i++)
    v[i] = i;
</script>
```

Quando si tratta di collezioni o vettori, è possibile scorrere tutti gli elementi facendo uso del costrutto ***for each***; esso è molto comodo perché lascia all'interprete l'onere di decidere quale sia il prossimo elemento e quando la collezione o il vettore ha esaurito gli elementi.

ESEMPIO

```
<script type="text/javascript">
  var v = ['s','e', 12, true];
  var s = "";
  for each(var elem in v)
    s = s + elem;
  alert(s);
</script>
```

6 Oggetti

In JavaScript non esiste il concetto di *classe*. Esiste, ed è ampiamente usato, il concetto di **oggetto**; il modo più semplice per creare un oggetto è usare la seguente forma

```
var oggetto = new Object();
```

ESEMPIO

```
<script type="text/javascript">
  var oggetto = new Object();
</script>
```

A questo punto è possibile definire un numero qualsiasi di **proprietà** (in altri linguaggi esse possono prendere il nome di *attributi*) facendo riferimento all'oggetto appena creato, seguito da un punto e dal nome della proprietà a cui assegnare un valore.

ES.

```
oggetto.descrizione = "Una descrizione";
oggetto.valore = 105;
```

In maniera molto simile si creano i **metodi**:

```
oggetto.toString = function() {this.descrizione};
```

OSSERVAZIONE La notazione che fa uso del carattere «punto» viene utilizzata anche per accedere al valore delle proprietà o per invocare i metodi: *nomeOggetto.nomeProprietà* o *nomeOggetto.nomeMetodo()*. A sua volta un oggetto potrebbe avere una proprietà che è anch'essa un oggetto; per accedere alle proprietà o ai metodi dell'oggetto più interno basta far riferimento all'oggetto principale, poi all'oggetto contenuto e, infine, al nome della proprietà: *oggetto.oggettoContenuto.proprietà*.

È molto comune creare metodi utilizzando funzioni senza nome (e infatti si chiamano **funzioni anonime**) al cui interno si fa riferimento a un particolare oggetto: `this`. Tale oggetto, come avviene in Java, altro non è che l'*oggetto corrente*, ovvero l'oggetto a cui si applica il metodo.
Anche in JavaScript, il metodo *toString* viene invocato per effettuare il *cast* dell'oggetto a stringa.

ESEMPIO

```
<script type="text/javascript">
var oggetto = new Object();
  oggetto.descrizione = "Una descrizione";
  oggetto.valore = 105;
  oggetto.toString = function(){return this.descrizione};
  alert(oggetto);
</script>
```

Mostra, come risultato, una finestra con il valore della proprietà descrizione.

6.1 Costruttori

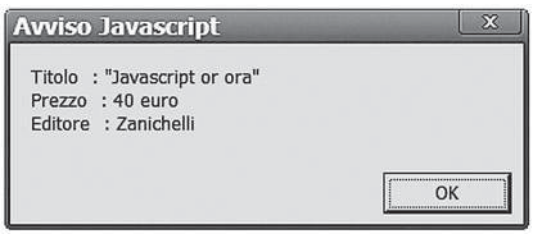
Quando una funzione accede a `this` e lo fa usando una *proprietà* in scrittura (ovvero a sinistra di un assegnamento), questo equivale a *creare* quella proprietà.

OSSERVAZIONE Un costruttore JavaScript è del tutto simile a una qualsiasi altra funzione. Si usa la stessa convenzione del Java secondo cui un costruttore ha sempre un nome con l'*iniziale maiuscola*, mentre il nome di una normale funzione inizia con una lettera minuscola.

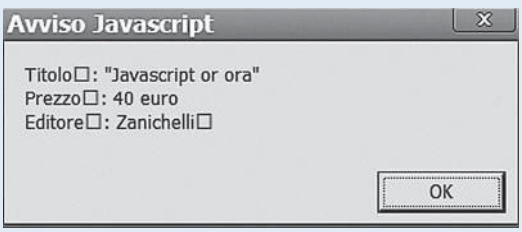
I **costruttori** creano nuovi oggetti e vengono scritti nella forma di una funzione al cui interno si creano tante proprietà quante sono necessarie. La creazione dell'oggetto avverrà invocando il costruttore preceduto dalla parola chiave **new**.

ESEMPIO

```
<script type="text/javascript">
function Libro(titolo, prezzo, editore){
  this.titolo = titolo;
  this.prezzo = prezzo;
  this.editore = editore;
  this.toString = function(){
    return 'Titolo\t: "'+this.titolo+'"\n'+
      'Prezzo\t: '+this.prezzo+' euro\n'+
      'Editore\t: '+this.editore+'\n';
  };
}

var mioLibro = new Libro('Javascript or ora', 40, 'Zanichelli');
alert(mioLibro);
</script>
```



OSSERVAZIONE Un caso emblematico di come browser diversi possano avere comportamenti diversi è la gestione dei caratteri speciali: alcuni di essi talvolta non riescono a gestire correttamente dei caratteri speciali o dei caratteri Unicode all'interno delle *alert*.



6.2 Oggetti come collezioni

In JavaScript un oggetto non è altro che una *collezione non ordinata* di proprietà.
A partire da JavaScript 1.2 è possibile usare la seguente notazione per la creazione di un oggetto:

```
var nomeOggetto = {elenco Proprietà};
```

ESEMPIO

```
<script type="text/javascript">
  var mc = {nome:'Maria', cognome:'Rossi', eta:3};
  var gm = {nome:'Giovanni', cognome:'Bianchi', eta:1};
</script>
```

Usando questa notazione, piuttosto compatta, anche la definizione di oggetti annidati diventa molto semplice.

ESEMPIO

```
<script type="text/javascript">
  var mc = {
    anagrafica: {nome:'Maria', cognome:'Rossi', eta:3},
    indirizzo:  {via:'dei mille', numero:100}
  };
</script>
```

6.3 Controllare l'accesso alle proprietà

Ciascuna proprietà può possedere attributi che permettono di gestire l'accesso alla proprietà stessa. I possibili attributi per ciascuna proprietà sono mostrati nella TABELLA 6.

TABELLA 6

Attributo	Significato
`Value`	Il valore della proprietà
`Writable`	Se `false` non è permessa la modifica alla proprietà `Value` (usando il *put*)
`Enumerable`	Se vale `true` questa proprietà comparirà anche nei cicli *for in*, altrimenti no
`Configurable`	Se a `false` qualsiasi tentativo di eliminare la proprietà o alterarne gli attributi (diversi da `Value`) fallisce

Per gestire questi attributi si ricorre al metodo *defineProperty* di *Object*. Esso crea una nuova proprietà se non esiste.

ESEMPIO

```
<script type="text/javascript">
  var persona = new Object();
  Object.defineProperty(persona, 'name',
    {
      value: 'Mario',
      writable: true,
      enumerable: true,
      configurable: true
    }
    );
  alert(persona.name);
</script>
```

Altrimenti è possibile usare il metodo *defineProperty* di *Object* anche per modificare gli attributi di una proprietà *esistente*.

ESEMPIO

Ecco come rendere una proprietà in sola lettura (*writable* a `false`), impedendo che qualcun altro modifichi tale impostazione (*configurable* a `false`):

```
<script type="text/javascript">
  var persona = new Object();
  persona.name = 'Mario';
  Object.defineProperty(persona, 'name',
    {
      writable: false, configurable: false
    } );
  alert(persona.name);
  persona.name = 'Giorgio'; // ignora l'assegnamento
  alert(persona.name);
</script>
```

6.4 Il prototipo di un oggetto

Quando si invoca un *costruttore*, o comunque si crea un oggetto con la parola chiave `new`, l'oggetto creato possiede una particolare proprietà chiamata `prototype` (*prototipo*) condivisa da tutti gli oggetti creati con quel costruttore. Come suggerisce il nome, questa proprietà serve a dare all'oggetto un *modello*, cioè qualcosa da cui partire per aggiungere altro. Qualsiasi proprietà o metodo aggiunto al prototipo viene automaticamente ereditato da tutti gli oggetti creati.

ESEMPIO

```
<script type="text/javascript">
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');

alert(mioLibro.sconto);   // mostra 10
alert(altroLibro.sconto);   // mostra 10

</script>
```

A ogni modo la regola è: se la proprietà (o il metodo) non sono definiti nell'oggetto, l'interprete li cerca nel prototipo. Pertanto se sono presenti due proprietà, una nell'oggetto e una nel prototipo, ha precedenza quella definita nell'oggetto.

ESEMPIO

```
<script type="text/javascript">
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;
alert(mioLibro.sconto); // mostra 20
</script>
```

È importante decidere quando creare una proprietà o un metodo dentro un oggetto o dentro il suo prototipo. Dato che se una proprietà o un metodo vengono definiti a livello di prototipo sono condivisi da tutti gli oggetti, all'interno di questo ha senso definirvi solo proprietà costanti, cioè quelle il cui valore non dipende dallo stato dell'oggetto. Per i metodi vale lo stesso ragionamento. Se riconsideriamo il costruttore *Libro* visto in precedenza, dove in esso si definiva il metodo *toString*, possiamo osservare che ha senso creare tale metodo a livello di prototipo.

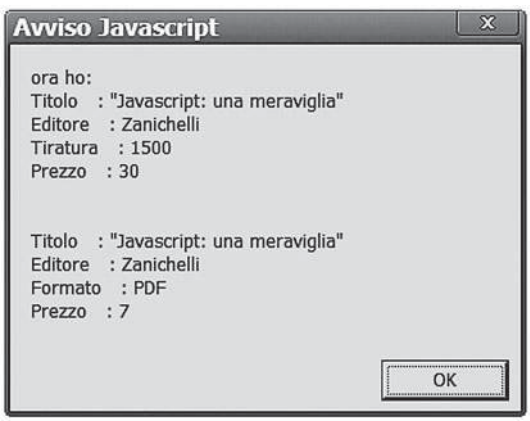
ESEMPIO

```
<script type="text/javascript">
function Libro(titolo, prezzo, editore){
  this.titolo = titolo;
  this.prezzo = prezzo;
  this.editore = editore;
}

Libro.prototype.toString = function(){
    return 'Titolo\t: "'+this.titolo+'"\n'+'Prezzo\t: '+this.prezzo+' euro\n'+
                    'Editore\t: '+this.editore+'\n';
  };

var mioLibro = new Libro('Javascript or ora', 40, 'Zanichelli');
alert(mioLibro);
</script>
```

Se anziché aggiungere metodi o proprietà, si ridefinisce il prototipo, si può ottenere un meccanismo simile all'estensione di un oggetto. Si consideri la classe `Libro` definita precedentemente; potremmo differenziare tra libri cartacei ed ebook usando oggetti diversi, ma che fanno uso dello stesso oggetto base.

ESEMPIO

```
<script type="text/javascript">
function Libro(titolo, editore){
  this.titolo = titolo;
  this.editore = editore;
  this.toString = function(){
    return 'Titolo\t: "'+this.titolo+'"\n'+'Editore\t: '+this.editore+'\n';
  };
}
var mioLibro = new Libro('Javascript: una meraviglia', 'Zanichelli');

function Stampa(libro, tiratura, prezzo){
  this.prototype = libro
  this.tiratura = tiratura;
  this.prezzo = prezzo;
  this.toString = function(){
    return this.prototype +'Tiratura\t: '+this.tiratura+'\n'+
      'Prezzo\t: '+this.prezzo+'\n';
  };
}
function EBook(libro, formato, prezzo){
  this.prototype = libro
  this.formato = formato;
  this.prezzo = prezzo;
  this.toString = function(){
    return this.prototype +'Formato\t: '+this.formato+'\n'+
      'Prezzo\t: '+this.prezzo+'\n';
  };
}
var mioLibroStampato = new Stampa(mioLibro, 1500, 30);
var mioLibroEBook = new EBook(mioLibro, 'PDF', 7);
alert('ora ho:\n'+ mioLibroStampato + '\n\n'+ mioLibroEBook);
</script>
```



6.5 Estendere i tipi predefiniti

L'utilizzo del prototipo è un meccanismo estremamente potente e flessibile, tanto che si potrebbe utilizzarlo per ridefinire i tipi di dato predefiniti del linguaggio.

ESEMPIO

Ecco come estendere l'oggetto *String* affinché esponga un nuovo metodo chiamato *palindroma* (che restituisce `true` quando la stringa letta da sinistra a destra o da destra a sinistra è uguale, come «etnagigante», `false` altrimenti):

```
<script type="text/javascript">
  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;
  };
  alert("etnagigante".palindroma()); // mostra true
</script>
```

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.

6.6 Oggetti come vettori associativi

Una qualunque proprietà di un oggetto può essere referenziata usando la sintassi *oggetto.proprietà*. La stessa cosa potrebbe essere fatta usando la seguente notazione: *oggetto['proprietà']*. In pratica è possibile accedere alle diverse proprietà usando una stringa che le identifica, in maniera del tutto analoga a quelli che vengono chiamati ***array* associativi**. Rispetto a quelli tradizionali, gli array associativi hanno un indice di tipo stringa anziché numerico. Questa situazione apre la strada a interessanti opportunità, per esempio componendo le stringhe di accesso alle proprietà in maniera dinamica.

ESEMPIO

```
<script type="text/javascript">
  var o = new Object();
  for(mese=1; mese<13; mese++)
    o['mese'+mese] = prompt('Dammi un valore per il mese '+mese);
  alert('il mese 5 il valore è '+o.mese5);
</script>
```

6.7 Funzioni innestate e *closure*

In JavaScript è possibile creare *funzioni innestate*, ovvero definite dentro altre funzioni. Esse hanno accesso a tutte le variabili visibili nel blocco dove sono definite; pertanto possono accedere a tutte le variabili globali, a quelle locali alla funzione, nonché ad altre funzioni innestate ma definite allo stesso livello della funzione presa in considerazione.

ESEMPIO

```
<script type="text/javascript">
function moltiplica(argomento1, argomento2){
   var fattore = argomento2;
      function moltiplicazione(cosa){
        return cosa * fattore;
      }
      return moltiplicazione(argomento1);
}
var valore = moltiplica(4, 7);
alert(valore);
</script>
```

A prima vista, nell'esempio non c'è nulla di particolare; infatti, come accade per le variabili definite dentro una funzione, anche le funzioni innestate sono visibili dentro il blocco che le racchiude. Pertanto l'utilizzo di funzioni innestate potrebbe sembrare piuttosto limitante. Si consideri però il seguente esempio, in cui la funzione innestata non viene usata, ma viene restituita fuori dalla funzione principale.

ESEMPIO

```
function moltiplicatore(argomento){
     function moltiplicazione(cosa){
       return cosa * argomento;
     }
     return moltiplicazione;
   }
var m = moltiplicatore(5);
alert(m(3));
```

Restituire una funzione pone il seguente problema: può accadere (come nell'esempio precedente) che quando viene *eseguita* la funzione, le variabili a cui essa aveva accesso in fase di *definizione* non esistano più in quanto, quando si esce da una funzione, le variabili locali dichiarate in essa vengono distrutte.
In realtà, se si prova a eseguire il codice sopra riportato, si può notare che l'esecuzione della funzione va a buon fine, mostrando come risultato 15.
Questo effetto è dovuto alle ***closure***. Le *closure* sono particolari oggetti che mantengono una *funzione* e l'*ambiente* in cui la funzione è stata definita.

Pertanto ogni volta che si crea una funzione innestata, viene creata una *closure*. Nell'esempio precedente ciò equivale alla creazione di una *closure* per la funzione innestata e alla creazione di un ambiente che contiene le variabili *cosa* e *argomento*. In pratica la *closure* contiene, nell'ambiente associato alla funzione, tutte le variabili visibili in quel momento.

ESEMPIO

```
<script type="text/javascript">
function applicaSconto(percentualeUlteriore){
  var percentualeBase = 10;
    function prezzoScontato(prezzo){
      var base = prezzo + prezzo * percentualeBase / 100;
      return base + base * percentualeUlteriore / 100;
    }
    return prezzoScontato;
}

var scontoNormale = applicaSconto(20);
var scontoSuper = applicaSconto(50);
var prezzoInizialeProdotto1 = 100;
alert(scontoNormale(prezzoInizialeProdotto1));
var prezzoInizialeProdotto2 = 100;
alert(scontoSuper(prezzoInizialeProdotto2));

</script>
```

In questo caso *applicaSconto* è un *generatore di funzioni* a cui applica uno sconto base del 10%, più uno sconto definito nel momento della sua invocazione. Nell'esempio sono create due funzioni che applicano, rispettivamente, uno sconto del 10% + 20% e uno di 10% + 50% (la seconda percentuale si applica al risultato della prima; si verifichi che è diverso rispetto ad applicare, rispettivamente, uno sconto del 30% e del 60% sul prezzo iniziale).

OSSERVAZIONE Quando, nel prossimo capitolo, verranno analizzate alcune applicazioni avanzate di JavaScript in ambito web, si vedrà come le *closure* siano estremamente utili in svariati contesti e, soprattutto, come **funzioni call back**, cioè funzioni che vengono «agganciate» ad alcuni oggetti (per esempio a un'immagine) e invocate al verificarsi di determinati eventi (per esempio quando l'immagine è stata caricata).

7 Oggetti predefiniti

In JavaScript ci sono oggetti predefiniti che facilitano la scrittura dei programmi grazie ai metodi e alle proprietà che espongono. Sono quelli presentati nella TABELLA **7**.

Oggetti predefiniti

Gli oggetti qui analizzati dovrebbero essere presenti in qualsiasi implementazione di JavaScript che segua correttamente le specifiche. Molti di questi oggetti evolvono insieme al linguaggio, presentando nuovi metodi e nuove proprietà. Purtroppo, se si assume di utilizzare, per esempio, un nuovo metodo, resta il problema della compatibilità con le versioni precedenti del linguaggio. Anche in questo caso aiuta il ricorso a una libreria esterna come jQuery, che si fa carico di implementare quelle funzionalità non rese disponibili a livello nativo dal linguaggio.

TABELLA 7

Oggetto	Utilizzo
`global`	Definisce funzioni di utilità e costruttori
`Object`	Creazione di oggetti e loro gestione
`Function`	Creazione e personalizzazione di funzioni
`Array`	Gestione di vettori
`String`	Gestione e manipolazione di stringhe
`Boolean`	Gestione dei valori di verità e loro conversioni
`Number`	Fornisce metodi per la gestione di numeri, insieme a costanti che identificano i valori minimi e massimi rappresentabili, costanti che identificano più e meno infinito e un particolare valore che è NaN (*Not-a-Number*)
`Math`	Costanti (come la costante *e*, π, …) e funzioni matematiche (di arrotondamento, funzioni goniometriche e così via)
`Date`	Gestione di date e orari
`RegExp`	Definizione e utilizzo di espressioni regolari
`JSON`	Gestione del JavaScript Object Notation, formato per la comunicazione tra un client JavaScript e un server
`Error`	Gestione degli errori, insieme agli oggetti EvalError, RangeError, ReferenceError, SyntaxError, TypeError e URIError

Di seguito verranno approfonditi gli oggetti predefiniti *Array*, *String* e *Date*.

7.1 *Array*

L'oggetto *Array* permette la gestione di vettori, ovvero di *insiemi ordinati* di elementi; ciascun elemento è recuperabile specificando l'indice, ovvero la sua posizione nel vettore (tale numero deve essere un numero positivo che parte da 0, prima posizione del vettore). La notazione per accedere a un elemento, sia in lettura sia in scrittura, è *nomeDelVettore[indice]*. Per *creare* un *Array* esistono tre costruttori:

- un costruttore senza parametri, per la creazione di un *Array* vuoto;
- un costruttore con numero di argomenti variabile, a cui passare come parametri gli elementi del vettore (separati da virgola);
- un costruttore con un unico argomento, un numero positivo, che indica il numero di elementi iniziali del vettore (in quest'ultimo caso tutti i valori degli elementi del vettore valgono *undefined*).

ESEMPIO

```
<script type="text/javascript">
  var v1 = new Array(); // costruisce un vettore vuoto
  var v2 = new Array(1.1, 'Una prova', 10); // costruisce un vettore di tre elementi,
                                            // inizializzandone il valore
  var v3 = new Array(10); // costruisce un vettore di 10 elementi con tutti i valori
                          // che valgono undefined
</script>
```

Ricordiamo che esiste anche la possibilità di creare un array partendo da una lista di elementi racchiusi tra parentesi quadre.

ESEMPIO

```
<script type="text/javascript">
  var vA = [1, 2, 3];
  var vB = ['primo', 2, 'terzo', , , 3];
  var vC = [[1, 2, 'test'], ['A', 'B']];
</script>
```

Si noti come il vettore *vB* contenga anche posizioni a cui non viene assegnato alcun valore. *vC* rappresenta un vettore multidimensionale; in particolare è un vettore a due dimensioni. Si noti come la seconda dimensione possa essere composta da un numero di elementi variabile.

L'oggetto *Array* possiede un certo numero di proprietà e metodi; i principali sono sintetizzati nella TABELLA **8**.

TABELLA **8**

Metodo	Significato
`concat`	Riceve come parametri un numero qualsiasi di array; restituisce un array i cui elementi sono tutti quelli dell'array a cui si applica il metodo, seguiti dagli elementi del primo array passato come argomento, poi quelli del secondo e così via.
`join`	Riceve come parametro una stringa (delimitatore) e restituisce una stringa formata dai singoli elementi dell'array separati dal delimitatore; se non viene passato alcun parametro usa il carattere virgola.
`pop`	Metodo senza argomenti: restituisce l'ultimo elemento dell'array, rimuovendolo da esso.
`push`	Riceve un numero qualsiasi di parametri e aumenta l'array inserendo i parametri in coda all'ultimo.
`slice`	Riceve due parametri, che sono l'indice di inizio e fine, restituendo un array che è la porzione di elementi che va da inizio (incluso) a fine (escluso); l'indice di fine può essere omesso e, in questo caso, restituisce tutti gli elementi a partire dall'inizio.
`sort`	Ordina gli elementi dell'array secondo l'ordine predefinito; se si vuole usare un ordine definito dall'utente, va passata come argomento una funzione che realizza il confronto fra due elementi.
`reverse`	Modifica l'array invertendo l'ordine degli elementi.

ESEMPIO

```
<script type="text/javascript">
  var vA = [1, 2, 3];
  alert(vA.join("; "));  // mostra "1; 2; 3"
  alert(vA.reverse());   // mostra "3,2,1"
  alert(vA.concat(vA));  // mostra "3,2,1,3,2,1"
</script>
```

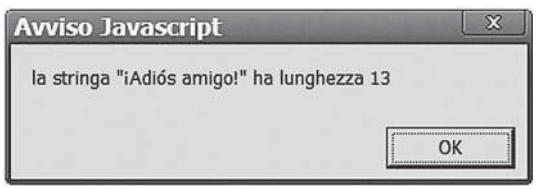
7.2 String

L'oggetto *String* permette la manipolazione di stringhe. Possiede due costruttori, uno senza parametri e uno con un parametro di tipo stringa; esso mette inoltre a disposizione la proprietà *length* che indica la lunghezza di una stringa e vari metodi di utilità. Tra di essi il metodo *charAt(posizione)* permette di recuperare il carattere presente alla posizione specificata, mentre *charCodeAt(posizione)* recupera il valore Unicode; il metodo *concat* restituisce una nuova stringa data dalla stringa a cui si applica il metodo concatenata con la stringa passata come parametro (*concat* può avere più di un parametro; se ce ne sono più di uno, concatena tutti i parametri alla stringa originaria).

ESEMPIO

```
<script type="text/javascript">
  var str = "\u00A1Adi\u00F3s amigo!";
  var msg = new String(
    'la stringa "'+str+'" ha lunghezza '+str.length+'\n');
  for(i=0; i<str.length; i++)
    msg.concat('\n', str.charAt(i), ' - ')
      .concat(str.charCodeAt(i));
  alert(msg);
</script>
```

Il risultato dell'esecuzione del codice sopra-riportato è quello che segue:



OSSERVAZIONE Usare un costruttore a cui passare una stringa oppure usare la sola stringa e assegnarla alla variabile sono strade di fatto equivalenti. Una, però, rende esplicito l'uso di un oggetto, mentre l'altro lo maschera. Si pensi anche a quanto sia più comodo l'uso dell'operatore di concatenazione «+» anziché il metodo *concat*. Di fatto l'uso delle stringhe è così frequente che le due operazioni fondamentali, la creazione e la concatenazione, hanno una notazione semplificata.

Altri metodi molto utilizzati di *String* sono *indexOf* e *substr*. *indexOf* permette di sapere se la stringa passata come parametro è contenuta nella stringa di partenza; se sì restituisce la posizione dove inizia, se no restituisce –1. *substr*, invece, restituisce una sottostringa: può avere uno o due parametri; se ce n'è uno restituisce la sottostringa che inizia dalla posizione specificata fino alla fine, se ce ne sono due restituisce la sottostringa che inizia

alla posizione specificata e fino al carattere la cui posizione è specificata nel secondo parametro (attenzione al fatto che il carattere di inizio è incluso nel risultato, quello di fine no).

ESEMPIO

```
<script type="text/javascript">
  var str = "Un esempio di stringa";
  var cosa = str.indexOf('per esempio'); // restituisce -1
  var pos = str.indexOf('esempio');      // restituisce 3
  var msg = str.substr(pos);             // restituisce 'esempio di stringa'
  var altro = str.substring(pos, 7);     //restituisce 'esem'
</script>
```

OSSERVAZIONE Quando si parla di posizioni di caratteri in una stringa si deve tener presente che, in JavaScript, il primo carattere è in posizione 0, il secondo in posizione 1 e così via, in maniera analoga a quello che vale per le posizioni degli elementi di un vettore.

7.3 *Date*

L'oggetto *Date* viene utilizzato per la gestione di date e di orari. Possiede due costruttori: il primo è senza parametri e costruisce un oggetto *Date* il cui contenuto è dato dalla data e ora impostata sul sistema al momento della creazione dell'oggetto. L'altro costruttore ha sei parametri che specificano, rispettivamente, l'anno, il mese, il giorno, le ore, i minuti, i secondi e i millisecondi. È possibile specificare anche solo anno e mese, sempre obbligatori, e fermarsi a un livello qualsiasi lasciando non espressi i valori degli altri parametri; in questo caso si assume che essi valgono zero.
Nella TABELLA 9 sono indicati i principali metodi.

TABELLA 9

Metodo	Significato
`getDate/setDate`	Legge/scrive il giorno del mese
`getMonth/setMonth`	Legge/scrive il mese
`getFullYear()/setFullYear()`	Legge/scrive l'anno
`getHours()/setHours()`	Legge/scrive l'ora
`GetMinutes/setMinutes`	Legge/scrive i minuti
`GetSeconds/setSeconds`	Legge/scrive i secondi
`GetMilliseconds()/setMilliseconds()`	Legge/scrive i millisecondi
`GetTime/setTime`	Legge/scrive una data, usando il formato del numero di millisecondi trascorsi dall'1 gennaio 1970.

ESEMPIO

Ecco come si potrebbe calcolare il numero di giorni della data odierna dall'inizio dell'anno:

```
<script type="text/javascript">
  var oggi = new Date(); // nb: il mese va da 0 a 11
                         // il giorno da 1 a 31!
  var inizio = new Date(oggi.getFullYear(), 0);
  var durata = oggi - inizio;
  var giorni = Math.ceil(durata / 1000 / 60 / 60 / 24)
  alert(giorni);
</script>
```

OSSERVAZIONE *Math.ceil* è un metodo che permette di arrotondare all'intero successivo un numero reale. *Math* è un oggetto che fornisce metodi e costanti di utilità matematica.

Sintesi

Applicazioni web dinamiche. Sono quei siti web che realizzano applicazioni utilizzando un qualche linguaggio di scripting o plug-in.

JavaScript. È il linguaggio di scripting per eccellenza e moltissime Applicazioni Web Dinamiche fanno uso di questa tecnologia.

AJAX. È quella tecnologia che permette di aggiornare i contenuti delle pagine web con dati provenienti dal server senza ricaricare tutta la pagina.

Lo standard di riferimento si chiama **ECMAScript**; esso specifica il comportamento atteso di tutti gli oggetti standard nonché tutte le regole che i diversi interpreti devono rispettare.

ECMAScript non definisce il comportamento atteso degli oggetti definiti dalle applicazioni host. Da qui nascono sottili **incompatibilità** tra browser diversi.

JavaScript possiede meccanismi che dovrebbero, a meno di errori di implementazione, garantire la **sicurezza** e l'impossibilità di creare virus o altro codice dannoso all'insaputa degli utenti.

JavaScript è un linguaggio interpretato, le cui variabili possono contenere qualunque tipo di dato riconosciuto da **linguaggio orientato agli oggetti**.

Il codice JavaScript può essere contenuto o in un **file esterno** o in un punto qualsiasi di una pagina HTML.

È possibile realizzare **conversioni esplicite**; in molti contesti il linguaggio crea delle **conversioni implicite**.

Ogni oggetto possiede una proprietà chiamata *prototype* (*prototipo*) che è condivisa con tutti gli altri oggetti di quel tipo.

Creare proprietà e metodi sul prototipo di un tipo equivale a renderli immediatamente disponibili a tutti gli oggetti di quel tipo.

Grazie al prototipo è possibile estendere anche i tipi predefiniti.

È possibile definire **array** di elementi **non omogenei** e a una o più dimensioni.

È possibile creare **nuovi oggetti**, definendo proprietà, metodi e costruttori. Un oggetto può essere visto come un **array associativo**, dove i nomi delle proprietà sono le chiavi dell'array. È possibile limitare l'accesso alle singole proprietà definendo opportuni attributi.

Il linguaggio permette di creare **funzioni innestate** e, grazie alle ***closures***, permette di restituirle e invocarle in posti diversi dalla loro definizione, preservando l'accesso a tutte le variabili di cui ha visibilità.

Tra i tipi predefiniti ci sono *Date*, per la gestione di date, *String* per la gestione delle stringhe e *Array* per la gestione dei vettori.

QUESITI

1 **È possibile verificare la correttezza dei dati, usando JavaScript, prima di inviarli al server?**

A Sì, sempre.
B No, mai.
C Dipende da quale linguaggio di programmazione viene usato sul server.

2 **I controlli lato server per la validazione dei dati devono essere fatti?**

A Dipende se c'è stata o no validazione sul client.
B Sì, sempre.
C No, mai.
D Dipende dai dati; se trattano dati «sensibili» sì, altrimenti no.

3 **Cosa si intende per effetto roll-over?**

A Significa che, seguendo un link, la pagina successiva va sopra la pagina precedente.
B Indica la sovrapposizione di testo e immagine per errori sulla pagina.
C Un effetto grafico per cui un'immagine cambia aspetto quando il cursore del mouse ci si posiziona sopra.
D Un'immagine statica ma dai colori molto vivaci.

4 **Cosa si intende per Web 2.0?**

A Il secondo rilascio delle specifiche dei protocolli web.
B Una modalità di concepire i siti web innovativa, focalizzata sulla collaborazione.
C È un modo per indicare un sito molto bello.
D Nessuna delle risposte precedenti.

5 **Cosa permette di fare la tecnologia AJAX?**

A Aggiornare una pagina ricaricandola completamente.
B Recuperare degli aggiornamenti sul server e riportarli sulla pagina senza ricaricarla completamente.
C Nessuna delle risposte precedenti.

6 **Che tipo di applicazioni sono GoogleDocs?**

A Foglio di calcolo, word processor e software per la creazione di presentazioni, tutte rigorosamente online.
B Ricerche su tutte le banche dati pubbliche.
C Giochi e cruciverba.
D Nessuna delle risposte precedenti.

7 **Quale azienda ha creato il linguaggio JavaScript?**

A Sun Microsystems.
B Microsoft.
C Netscape.
D IBM.

8 **La «Legge Stanca» si deve applicare, a meno di sanzioni, a …**

A … tutti i siti.
B … solo ai siti di pubbliche amministrazioni rivolte al pubblico.
C … a tutti i siti purché non siano commerciali.
D … a nessun sito; è solo una raccomandazione.

9 **Quali delle seguenti tecnologie necessitano di appositi plug-in per essere eseguite?**

A Applet Java.
B Codice JavaScript.
C HTML.
D SilverLight.

10 **Come si rappresentano i caratteri Unicode in JavaScript?**

A \xNNNN
B \uNNNN
C #refNNNN
D Nessuna delle risposte precedenti.

11 **Quali sono i caratteri che permettono di inserire commenti su più righe?**

A /* */
B //
C /# #/
D Nessuna delle risposte precedenti.

12 Che valore assume una variabile dichiarata ma non inizializzata?

A 0.
B NaN.
C *undefined*.
D Nessuna delle risposte precedenti.

13 Quali dei seguenti numeri sono rappresentazioni corrette in base 16?

A 0x22.
B 0X22.
C 0u22.
D 0U22.

14 Quanto vale la valutazione dell'espressione `true && false || true`?

A `true`
B `false`
C Nessuna delle risposte precedenti; c'è un errore nell'espressione.

15 Quanto vale la valutazione dell'espressione 10=="10"?

A `true`
B `false`
C Nessuna delle risposte precedenti; c'è un errore nell'espressione.

16 Se *x* vale 10, dire il valore di *x* e *y* dopo che viene eseguita la seguente istruzione: *y* = *x*++.

A $x = 10; y = 10$.
B $x = 11; y = 10$.
C $x = 10; y = 11$.
D $x = 11; y = 11$.

17 Quanto vale la valutazione dell'espressione 1*2 + "a"+3*2+5?

A 2a11.
B 2a65.
C 12a325.
D Nessuna delle risposte precedenti.

18 Quali delle seguenti stringhe è sintatticamente corretta?

A "Un'idea"
B 'Un\'idea'
C 'Un"idea'
D Tutte le risposte precedenti.

19 Quanto vale [3,'d','o'].join('/') ?

A '3do'
B '3,d,o'
C '33ddoo'
D Nessuna delle risposte precedenti.

20 Se il metodo *getTime* di un oggetto *Date* restituisce 0, esso rappresenta …

A … il giorno 0 d.C.
B … il 1 gennaio 1970.
C Non può mai restituire 0.
D Significa che la data non è stata inizializzata.

21 È possibile aggiungere un metodo all'oggetto predefinito *Date*?

A No, essendo predefinito non si può modificare.
B Sì, basta ridefinire il suo costruttore.
C Sì, basta aggiungere il metodo al suo prototipo.
D Nessuna delle risposte precedenti.

ESERCIZI

1 Quando si usano librerie di terze parti è normale che esse vengano incluse come file .js separati dal resto delle proprie pagine e dei propri file JavaScript. È possibile scaricare la libreria e distribuirla insieme alle proprie pagine e referenziarla, al loro interno, con URL relative. Ultimamente è pratica comune quella di non includerla nelle proprie pagine, ma di referenziarla sul sito ufficiale che distribuisce la libreria con una URL assoluta. Quali vantaggi si riscontrano in una simile pratica?

2 Si consideri la seguente pagina HTML contenente una form, due campi di inserimento dati e un

terzo campo in sola visualizzazione; si associno ai quattro pulsanti gli opportuni gestori JavaScript affinché, quando si preme su uno di essi, vengano letti i valori dei primi due campi, sia eseguita l'operazione aritmetica corrispondente al tasto premuto e sia visualizzato il risultato sul terzo campo.

```
<html>
  <body>
    <form>
      <input type="text" id="val1" />
      <input type="button"
        id="piu" value="+" />
      <input type="button"
        id="meno" value="-" />
      <input type="button"
        id="diviso" value="/" />
      <input type="button"
        id="per" value="*" />
      <input type="text" id="val2" />
      =
      <input type="text"
        id="ris" readonly="readonly" />
    </form>
  </body>
</html>
```

3 Dato il seguente script, cercare di comprendere cosa viene visualizzato (si consiglia la sua esecuzione all'interno di un browser solo dopo una simulazione «a mano»):

```
<script type="text/javascript">
  var a = 1;
  function funzione(){
    var b = 2;
    c = 3;
  }
  funzione();
  alert(a + c);
  alert(a + b);
</script>
```

4 Creare una funzione che restituisce il valore minimo di un vettore di numeri passato come argomento.

5 Rivedere l'esercizio precedente e chiedersi cosa accade se, anziché un vettore di numeri, viene passato un vettore di stringhe.

6 Creare un programma che, specificato un numero, crea una stringa formata da tutte le ultime cifre dei numeri che vanno da 0 al numero specificato; esempio; 13 → 01234567890123.

7 Creare un oggetto, *Rettangolo*, il cui costruttore riceve due parametri che rappresentano ciascuno, rispettivamente, l'altezza e la larghezza e imposta due nuove proprietà che memorizzano tali dati.

8 Estendere l'oggetto del punto precedente facendo sì che abbia anche un metodo chiamato *area* che restituisce l'area del *Rettangolo*.

9 Considerare il metodo *area* del punto precedente. Dove è più appropriato definirlo? Dentro il costruttore come metodo dell'oggetto oppure come metodo del prototipo?

10 Creare un oggetto *Campionato* con al suo interno un vettore di squadre; ciascuna squadra sarà un oggetto con attributi *nome*, *punti* e *colore maglia*. Far sì che quando si stampa a video (per esempio con una *alert*) un oggetto di tipo *Campionato*, vengano mostrati tutti i nomi delle squadre e relativo punteggio, ognuna su una riga diversa dalla precedente.

11 Fare in modo che gli oggetti di tipo *String* possiedano un nuovo metodo, che è *vocali*, che restituisce solo le vocali della stringa di origine.

12 Sapendo che l'oggetto *Date* possiede, tra gli altri, i seguenti costruttori:

- *new Date*()
- *new Date*(anno, mese, giorno, ore, minuti, secondi, millisecondi)

dove il primo restituisce la data dell'orologio del sistema e il secondo imposta una data a partire dai suoi componenti (quando non specificati valgono zero, per esempio *new Date*(2000, 1, 1) impostare il 1° gennaio del 2000 alle ore 00:00:00), creare un oggetto *Persona* con un costruttore che riceve tre parametri: nome, cognome e data di nascita. Tale costruttore dovrà creare le proprietà *nome*, *cognome* ed *età* (quest'ultima la si può calcolare a partire dalla data dell'orologio del sistema).

B2 JavaScript e il DOM, jQuery e Google *Maps*

1 Il browser come ambiente di esecuzione

Nel capitolo precedente gli script d'esempio non facevano alcuna assunzione sull'ambiente di esecuzione (l'unica eccezione era rappresentata dalle funzioni di I/O verso l'utente – *alert* e *prompt* – che sono funzionalità offerte dai browser). Questo fa sì che tali script possano essere eseguiti in diversi contesti. La loro generalità si scontra con un limite: non potendo far riferimento a oggetti «esterni», sono di poca o scarsa utilità per sviluppare vere e proprie applicazioni. In questo capitolo si supererà anche questo limite facendo riferimento esplicitamente ai browser per la creazione di applicazioni web e creando script, anche complessi, che traggono vantaggio dagli elementi di pagine HTML e delle funzionalità offerte dai browser.

1.1 Pagine web rappresentate con alberi

I tag `<script>`, contenenti del codice JavaScript, possono essere inseriti in qualsiasi punto di una pagina web e, volendo, possono concorrere a crearne il contenuto.

DHTML

Il W3C definisce il DHTML come: «Dynamic HTML (DHTML) is a term used by some vendors to describe the combination of HTML, style sheets and scripts that allows documents to be animated».
In pratica ci si riferisce alle tecnologie che permettono di variare l'aspetto (e i contenuti) di una pagina una volta che essa è stata caricata e visualizzata su un browser. JavaScript è la tecnologia per eccellenza che rende possibile tale dinamicità. In questo capitolo si descrive come ottenere tale risultato utilizzando API e librerie appropriate, unitamente all'analisi di alcuni problemi legati alle incompatibilità fra browser diversi.

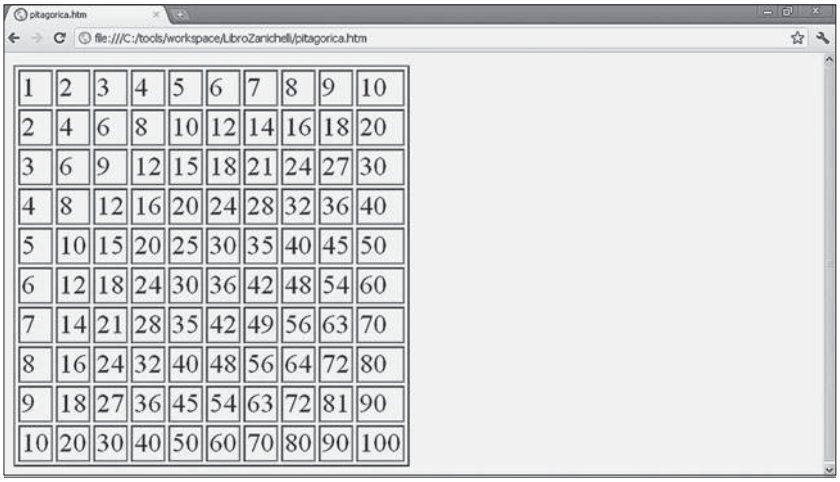
ESEMPIO

Ecco una pagina che crea una pagina web contenente una tavola pitagorica; per la formattazione usa una tabella di 10 righe e 10 colonne. Ci sono due cicli nidificati: quello più interno genera una riga intera, quello più esterno apre e chiude una nuova riga. La FIGURA **1** mostra l'output che esso produce.

```
<html>
<body>
<script type="text/javascript">
  var r = '<table border="1">';
  for(var i=1; i<11; i++){
    r = r + '<tr>';
    for(var j=1; j<11; j++){
      r = r + '<td>' + i*j + '</td>';
    }
    r = r + '</tr>';
```

▶

```
    }
    r = r + '</table>';
    document.write(r);
</script>

</body>
</html>
```



1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

FIGURA **1**

Motori di rendering

Ogni browser è dotato di un componente che si occupa del rendering delle pagine HTML; tale componente viene spesso indicato come *motore di rendering*.

Alcune aziende adottano soluzioni proprietarie, cioè sviluppate da sé e il cui sorgente non è pubblico; altre hanno progetti open source sviluppati da zero, altre ancora hanno adottato motori di rendering ancora open source ma sviluppati da terze parti.

Per tutti i progetti open source è possibile eseguire il download dei sorgenti e analizzarli; questo aiuta, tra le altre cose, a comprendere quali strutture dati siano usate per la rappresentazione dei documenti HTML e con quali regole sono popolate.

L'istruzione `document.write` crea un nuovo contenuto della pagina; tale contenuto viene creato nella posizione dove compare lo script. Il contenuto, nel caso si faccia uso di `document.write`, deve essere fornito come stringa che rappresenta porzioni complete di tag HTML; ciascuna stringa viene interpretata dal browser per la sua visualizzazione.

In realtà i browser interpretano una pagina HTML e creano una struttura interna per la sua rappresentazione; tale struttura è un albero, dove i diversi nodi sono i tag da rappresentare, partendo dal padre che è il tag HTML, passando dai suoi figli, normalmente i tag `<head>` e `<body>`, e così via, dove ciascun figlio altro non è che un tag contenuto dal padre. Affinché ciascun browser possa creare la struttura ad albero in modo univoco, la pagina HTML deve essere scritta in maniera corretta: ogni tag aperto deve essere chiuso, tag diversi possono essere contenuti uno dentro l'altro ma mai sovrapposti (ovvero non è possibile aprire un tag, poi un altro, chiudere il primo e solo dopo il secondo).

In generale, qualunque documento HTML può essere rappresentato come un albero dove gli elementi sono tutti raggiungibili dalla radice-body. I nodi di tale albero, e di conseguenza ciascun elemento della pagina, possono essere raggiunti, in lettura o in scrittura, attraverso l'uso di opportune **API** dal browser. In questo modo è possibile leggere il contenuto dell'albero, ma anche modificarlo o creare nuovi nodi. Le API esposte dal browser sono oggetti, ciascuno dei quali modella specifiche proprietà e comportamenti delle diverse parti di una pagina. Tale modello prende il nome di **DOM** (*Document Object Model*).

OSSERVAZIONE Il DOM è il modo con cui un browser espone un documento HTML, indipendentemente dalla tecnologia che vi accede. In questo senso esso non è strettamente legato a JavaScript, anche se il JavaScript rappresenta il linguaggio di programmazione maggiormente usato per accedervi.

ESEMPIO

Validare una pagina HTML significa verificare che rispetti lo standard. Una pagina non è valida, per esempio, quando un certo tag è aperto ma non chiuso:

```
<p>esempio<br/>
<p>test</p>
```

Tale frammento si potrebbe interpretare come:

```
<p>test</p><br/>
<p>test</p>
```

oppure come:

```
<p>test<br/>
<p>test</p></p>
```

In base all'interpretazione è chiaro che il frammento di codice HTML assume significati diversi, generando rappresentazioni interne diverse e, in ultima analisi, può essere fonte di incompatibilità tra browser diversi. Ecco perché è raccomandato scrivere sempre codice HTML valido, anche se, in molti casi, i browser interpretano comunque correttamente anche documenti non validi.

OSSERVAZIONE Esistono, per fortuna, strumenti automatici per la validazione delle pagine. Tra i servizi più utilizzati c'è quello del W3C (http//validator.w3.org/unicorn). Esistono poi numerosi plug-in da installare sui browser affinché la validazione possa essere fatta già dal browser mentre visita le diverse pagine.

W3C e il DOM

Il W3C è una comunità internazionale che si occupa di standard per il web. Tra i vari standard definisce anche quali sono le API messe a disposizione dai browser e, in particolare, quelle che permettono di accedere e gestire il DOM.
Esistono vari standard suddivisi in livelli: DOM Level 1 definisce la navigazione all'interno di documenti HTML e XML e la modifica del loro contenuto.
Il Level 2 estende il Level 1 per il supporto dei namespace XML, la manipolazione dei CSS, la gestione degli eventi e così via. Sia il Level 1 sia il Level 2 sono W3C Recommendations: questo significa che lo standard è considerato definitivo e consolidato.
Il Level 3 è ancora in una versione soggetta a cambiamento, chiamata Working Draft. Il Level 3 consiste in 6 specifiche differenti: le funzioni di base (nucleo); il caricamento e salvataggio; l'uso di Xpath; viste e formattazione; requisiti e, infine, validazione.

1.2 Accedere ai nodi di un documento

L'oggetto `document` permette l'accesso all'albero che rappresenta il documento; la radice, rappresentata dal tag `<html>`, è accessibile dalla proprietà `document.documentElement`. Per accedere a nodi specifici è possibile riferirsi al loro (eventuale) ID e recuperarli grazie al metodo `getElementById`. Se si vogliono recuperare tutti i tag di un certo tipo, per esempio tutti i paragrafi `<p>`, si può ricorrere al metodo `getElementByTagName`, passando come argomento il nome del tag: `document.getElementByTagName('p')`. Il metodo restituisce un vettore, dove ciascun elemento è un nodo del tipo cercato; pertanto, per riferirsi a uno specifico elemento, è necessario conoscere la sua posizione nel vettore restituito.

ESEMPIO

Ecco, per esempio, come mostrare l'attributo `name` di un elemento recuperato sia grazie al suo ID, facendo uso del metodo `getElementById`, sia grazie al suo tipo di tag e la posizione all'interno del vettore di elementi restituito dal metodo `getElementsByTagName`:

```
<html>
<body>
  <input type="button" id="a" name="aaa" />
  <input type="button" id="b" name="bbb" />
  <input type="button" id="c" name="ccc" />
  <input type="button" id="d" name="ddd" />
<script type="text/javascript">
  alert(document.getElementById("a").name); //mostra aaa
  alert(document.getElementsByTagName("input")[0].name); //mostra aaa
</script>
</body>
</html>
```

OSSERVAZIONE L'albero di un documento HTML viene costruito man mano che il browser legge i diversi tag della pagina. Che accade se una parte del codice JavaScript tenta di accedere a parti del documento HTML che seguono, ovvero di cui non è ancora stato costruito l'albero? È evidente che si presenta un errore o comunque, in dipendenza dall'operazione fatta, lo script non risponde come dovrebbe. Che accade se una porzione di programma JavaScript tenta di accedere a parte del documento che precede lo script, ma in un momento in cui non è ancora caricata l'intera pagina?
Non esiste alcuna risposta sempre valida; anche, empiricamente, si può verificare che funziona con gran parte dei browser, non è garantito che sia sempre così. L'unico modo per essere sicuri di poter accedere al DOM senza errori è aspettare che tutto il documento venga caricato; a tal fine è possibile fare uso del gestore di evento `onload` sul tag `<body>`, che identifica proprio l'evento del caricamento completo della pagina attuale e, di conseguenza, la certezza che la rappresentazione interna del documento è completa.

ESEMPIO

In questo esempio di fa uso del metodo `getElementById` per recuperare un elemento del DOM fornendo il suo ID. Nel primo caso il metodo va a buon fine, perché l'elemento a cui si tenta di riferirsi è già stato creato; nel secondo caso non restituisce l'oggetto, perché esso deve ancora essere costruito.

```
<html>
<body>
  <input type="button" id="b" name="b" />
<script type="text/javascript">
  alert(document.getElementById("b")); // mostra un oggetto
  alert(document.getElementById("c")); // mostra null
</script>
  <input type="button" id="c" name="c" />
</body>
</html>
```

Anche nel primo caso, il modo corretto per accedere in lettura all'elemento del DOM sarebbe quello di farlo dopo il caricamento di tutta la pagina:

```
<html>
<body onload="alert(document.getElementById('b'))">
  <input type="button" id="b" name="b" />
  <input type="button" id="c" name="c" />
</body>
</html>
```

OSSERVAZIONE Esiste anche il metodo `getElementsByName` che, come si può intuire, restituisce tutti gli elementi che hanno un certo valore nell'attributo `name`. A differenza dell'ID, che deve essere univoco, `name` può non esserlo. Ecco perché anche questo metodo restituisce un vettore di elementi (anche se di solito restituisce un solo elemento).

1.3 Aggiungere nodi a un documento

Esistono due metodi, chiamarti `createElement` e `appendChild`, utili a questo scopo: il primo crea un nuovo nodo, il secondo aggiunge a un elemento esistente un nodo, passato come argomento, di cui quest'ultimo diventa un figlio. Il metodo `createTextNode` permette la creazione di testo da aggiungere a un qualsiasi elemento (anche in questo caso un nodo di testo è creato come nodo a sé stante e poi viene aggiunto come figlio a un nodo che, per esempio, è un `<td>`).

ESEMPIO

Utilizzando i metodi `createElement` e `appendChild`, è possibile riscrivere l'esempio visto in precedenza per la creazione di una tavola pitagorica. Benché il risultato sia sempre lo stesso, questa porzione di codice rende molto più esplicito il modo con cui la pagina viene rappresentata internamente dal browser:

```
<html>
<body></body>
<script type="text/javascript">
  var table = document.createElement('table');
  table.setAttribute('border', '1');
  for(var i=1; i<11; i++){
    var row = document.createElement('tr');
    for(var j=1; j<11; j++){
      var col = document.createElement('td');
      var txt = document.createTextNode(''+ i*j);
      col.appendChild(txt);
      row.appendChild(col);
    }
    table.appendChild(row);
  }
  document.body.appendChild(table);
</script>
</html>
```

In FIGURA **2** il risultato della costruzione «ad albero» dell'esempio precedente.

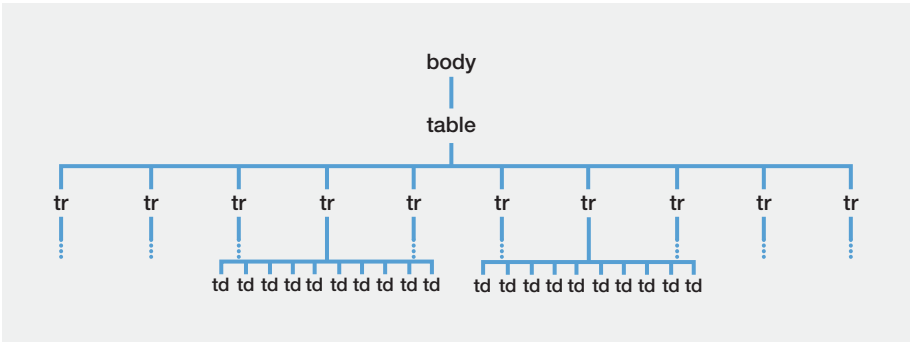


FIGURA **2**

Negli esempi appena illustrati, l'istruzione `document.write` altro non è che un metodo (`write`) esposto dall'oggetto `document`. Conoscere il DOM significa conoscere questo e altri oggetti esposti dai browser e le loro proprietà e i loro metodi.

1.4 Eventi associati alle pagine

Oltre all'evento `load` (gestore `onload`) è possibile associare del codice JavaScript al verificarsi di ulteriori eventi. Per specificare un gestore di evento lo si può definire come nome di attributo del tag a cui si applica; il nome del gestore è dato da due parti: un prefisso «on» e il nome dell'evento. Per esempio `onclick` è il gestore dell'evento `click`. In inglese si potrebbe tradurre con «sul click», che sta a significare, in maniera sottintesa, «al verificarsi dell'evento click». `onclick` si può applicare a vari elementi di una pagina; se lo si applica a un bottone, per esempio, esso verrà usato quando l'utente fa click sul bottone stesso. Come valore del tag si può associare una porzione di codice JavaScript.

ESEMPIO

Nulla vieta di inserire il codice JavaScript che gestisce un evento direttamente dentro il valore dell'attributo; ecco, per esempio, uno script che conta il numero di clic su un pulsante:

```
<html>
<script type="text/javascript">
  var i = 0;
</script>
<body>
  <input type="button" id="b" name="b" onclick="alert(++i)"/>
</body>
</html>
```

Ma questa soluzione è sensata solo per script molto semplici: è comune che il gestore invochi delle funzioni che racchiudono, al loro interno, la logica del gestore; ecco l'esempio precedente riscritto secondo tale convenzione: ▶

```
<html>
<script type="text/javascript">
  var i = 0;
  function mostraClick(){
    alert(++i);
  }
</script>
<body>
  <input type="button" id="b" name="b" onclick="mostraClick()"/>
</body>
</html>
```

Gli eventi gestibili sono molti. Nelle TABELLE **1** e **2** sono illustrati alcuni tra i più significativi eventi definiti nel documento del W3C chiamato *DOM Level 3 Events Specification*. Non vi compaiono gli eventi legati alla manipolazione o all'accesso al DOM, che servono per gestire le notifiche alla struttura del documento, perché il loro uso è, nel documento stesso, deprecato.

TABELLA **1**

Evento	Significato
`abort`	Il caricamento dell'immagine è stato interrotto
`blur`	L'elemento ha perso il focus
`click`	È stato fatto click sull'elemento
`dblclick`	È stato fatto doppio click sull'elemento
`error`	Errore durante il caricamento (del documento o dell'immagine)
`focus`	L'elemento ha ricevuto il focus
`keydown`	Un pulsante è stato premuto
`keypress`	Un pulsante è stato inviato
`keyup`	Un pulsante è stato rilasciato
`load`	Quando viene caricato un documento (HTML, CSS, JavaScript e così via)
`resize`	Finestra o frame ridimensionati
`scroll`	Quando una finestra o un altro elemento dotato di barre laterali subisce uno scroll
`select`	Selezione del testo
`textinput`	Evento a fronte dell'immissione di un testo (sia si tratti di immissioni da tastiera, da copia e incolla o altre operazioni simili)
`unload`	Uscita dalla pagina (chiusura del browser o cambio indirizzo)
`wheel`	Quando avviene uno spostamento della rotellina del mouse

Gli eventi che permettono di gestire l'interazione della pagina con il mouse sono riportati nella TABELLA **2**.

TABELLA **2**

Evento	Significato
`mousedown`	Un pulsante del mouse viene premuto
`mouseenter`	Quando un dispositivo di puntamento entra nell'area attiva dell'elemento o di uno dei suoi discendenti
`mouseleave`	Quando un dispositivo di puntamento esce nell'area attiva dell'elemento o di uno dei suoi discendenti
`mousemove`	Il cursore del mouse si è spostato
`mouseout`	Il cursore del mouse ha abbandonato l'elemento
`mouseover`	Cursore del mouse sopra l'elemento
`mouseup`	Un pulsante del mouse viene rilasciato

A complicare le cose c'è il fatto che vari browser supportano eventi che non fanno parte delle specifiche W3C. Per essi ci sono eventi ampiamente supportati, come `contextmenu`, l'evento che viene sollevato quando il browser dovrebbe mostrare il menu contestuale (per esempio quando l'utente fa click con il bottone destro sul documento), mentre altri eventi, benché facciano parte delle specifiche, sono implementati da pochi browser (un esempio in questo senso è costituito dell'evento `mouseenter`, che non sempre è supportato da tutti i browser).

OSSERVAZIONE La gestione degli eventi può presentare problemi legati alle diverse implementazioni dei browser. I modelli di riferimento W3C sono il *Document Object Model (DOM) Level 2 Events Specification* e la versione successiva, il *Document Object Model Level 3 Specification*.

OSSERVAZIONE Il W3C indica nel documento *DOM Level 3 Core Specification* che i browser devono esporre il seguente metodo:

```
document.implementation.hasFeature(feature, ver).
```

Esso restituisce `true` se gestisce la caratteristica *feature* nella versione *ver*, `false` altrimenti. Per esempio l'invocazione:

```
document.implementation.hasFeature("Events", "3.0")
```

restituisce `true` se supporta *DOM Level 3 Events Specification*.

ESEMPIO

Mentre l'evento `keyup`, che si verifica quando un tasto è stato rilasciato, è piuttosto intuitivo, potrebbero destare dei dubbi gli eventi `keydown` e `keypress`. Per comprenderne il significato si pensi che l'evento `keypress` avviene all'inserimento di un carattere. Per esempio, si consideri di tener premuto il tasto della lettera `'A'` per alcuni secondi; non appena lo si preme si verificano sia l'evento `keydown`, perché il tasto va dalla posizione normale a quella in basso, sia `keypress`, perché la lettera `'A'` viene inviata al programma. Ma tenendo premuto il tasto non avvengono altri eventi `keydown`, mentre verranno inviati via via un certo numero di caratteri `'A'`: a ogni invio viene sollevato un evento `keypress`. Ecco un esempio che illustra quanto detto, dove è possibile inserire dei caratteri su un campo di inserimento testo tenendo premuto un pulsante e verificando che quanto detto poc'anzi è vero: ▶

Browser e incompatibilità

Testare da sé le diverse caratteristiche e le specifiche è un lavoro impegnativo e che non si esaurisce nel tempo, in quanto versioni di browser sempre nuove e con particolarità specifiche vedono la luce. Il web è, anche in questo caso, una miniera ricca di informazioni ed esempi.

Esistono siti che offrono varie informazioni, per esempio tabelle in cui vengono sintetizzati i risultati dei test di compatibilità dei diversi browser rispetto agli standard W3C, sia per quanto concerne i CSS che le funzionalità JavaScript.

Talvolta sono presentati anche i sorgenti dei test eseguiti, che possono essere spunto per approfondimenti e ulteriori indagini svolte in autonomia.

```
<html>
<head>
<script type="text/javascript">
var i = 0;

function keyDown(){
  document.getElementById('b1').value=" DOWN ";
}

function keyUp(){
  document.getElementById('b1').value=" UP ";
}

function conta(){
  document.getElementById('b2').value= ++i;
}

</script>
</head>
<body>
<input id="campo" type="text" onkeydown="keyDown()"
  onkeyup="keyUp()" onkeypress="conta()"/>
<input id="b1" type="button" value=" " />
<input id="b2" type="button" value=" " />
</body>
</html>
```

2 DOM e gli oggetti esposti dal browser

Il DOM è una collezione di oggetti che modellano vari aspetti di una pagina. È importante osservare che tutti gli oggetti esposti nel DOM possiedono alcune proprietà in comune. Le principali proprietà condivise sono sintetizzate nella TABELLA 3.

TABELLA 3

Proprietà	Significato
`id`	Identificativo (dovrebbe essere univoco)
`attributes`	Restituisce un vettore di attributi che caratterizzano l'elemento
`childNodes`	Restituisce un vettore degli elementi (nodi) figli dell'elemento preso in considerazione (si ricorda la struttura ad albero di ciascun documento HTML; considerare i figli di un elemento equivale a considerare gli elementi contenuti nello stesso)
`className`	Proprietà in lettura e scrittura che identifica la classe di appartenenza dell'elemento
`innerHTML`	Restituisce la parte di codice HTML che comprende il tag stesso e il suo contenuto ▶

▶ TABELLA 3

Proprietà	Significato
nodeName	Nome del nodo corrente
nodeType	Tipo del nodo corrente
nodeValue	Valore del nodo corrente
parentNode	Il nodo padre rispetto al nodo corrente
style	Restituisce lo stile che si applica all'elemento corrente
tabIndex	Ordine dell'elemento per la navigazione tramite tasto di tabulazione

ESEMPIO

A questo punto dovrebbe essere più chiaro l'esempio del capitolo precedente, dove si realizzavano effetti rollover sulle immagini; in particolare ecco ripresentato uno degli script:

```
  <img src="prova1.gif"
  onmouseover="document.getElementById('txt').innerHTML='spiegazione primo elemento'" />
<br/>
<img src="prova1.gif"
  onmouseover="document.getElementById('txt').innerHTML='spiegazione secondo elemento'" />
<br/>
<img src="prova1.gif"
  onmouseover="document.getElementById('txt').innerHTML='spiegazione terzo elemento'" />
<br/>
<div id="txt"></div>
```

Ciascun gestore altro non fa che recuperare l'elemento con ID a `'txt'` (che è un `div`) e inserirci, come codice HTML, una scritta che varia in base all'elemento dove è posizionato il cursore del mouse.

ESEMPIO

L'esempio che segue realizza una funzione, chiamata *info*, che mostra alcune tra le informazioni «generali» comuni a tutti gli oggetti esposti dal DOM. Tale funzione viene usata per due elementi: body e un button. Si può modificare l'esempio per eseguire test su altri elementi del documento.

```
<html>
<head>
<script type="text/javascript">
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: "+info(obj.parentNode)+"<br/>";
    return s;
    }
}
</script>
</head>
<body>
  <input type="button" id="b" name="b" />
<script type="text/javascript">
```

▶

```
    window.document.write(
      info(document.getElementsByTagName("body")[0])
    );
    window.document.write(
      info(document.getElementById("b"))
    );
</script>
</body>
</html>
```

Oltre alle proprietà sopracitate, esistono anche una serie di metodi «generali». I più usati sono mostrati nella TABELLA 4.

TABELLA 4

Metodo	Significato
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

Nel seguito del capitolo verranno 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. Per esplorare le proprietà e i metodi enumerabili di un oggetto, è possibile usare la seguente funzione generale:

ESEMPIO

```
<html>
<script type="text/javascript">
function mostra(objName){
  for(prop in eval(objName)){
    var value=eval(objName+"['"+prop+"']");
    document.write("<b>"+prop + "</b>:" + value +"<br/>");
  }
}
</script>
<body onload="mostra('document')">
</body>
</html>
```

Attenzione: la funzione mostra solo le proprietà e i metodi enumerabili. Un tipo di browser potrebbe avere alcune proprietà enumerabili, un altro averle non numerabili. Come pure un browser potrebbe avere una proprietà (enumerabile o meno) e un altro non averla affatto. Pertanto questo è un metodo di indagine utile, ma non esaustivo.

Questa funzione è estremamente utile per indagare sulle diversità di implementazioni tra browser diversi, e verrà utilizzata per esplorare alcuni tra gli oggetti esposti (nell'esempio si usa `'document'` come argomento alla funzione `mostra`; al posto di `'document'` si potrà usare il nome dell'oggetto da esplorare).

OSSERVAZIONE La funzione `mostra` fa uso della funzione `eval`. Tale funzione è estremamente flessibile, in quanto permette di ricevere come argomento una stringa qualsiasi e «valutarla», ovvero interpretarla come codice JavaScript, sia esso invocazione a funzioni o uso di operatori.

ESEMPIO

Ecco come una pagina HTML con una tabella possa essere manipolata colorando lo sfondo delle righe in posizione dispari. L'esempio fa uso del metodo `getElementsByTagName` per recuperare tutte le righe (tag *tr*), e poi del metodo `setAttribute` per impostare il colore di fondo:

```
<html>
<body>
  <table border="1">
    <tr><td>Esempio</td></tr>
    <tr><td>Esempio</td></tr>
    <tr><td>Esempio</td></tr>
  </table>
<script type="text/javascript">
  var td = document.body.getElementsByTagName('tr');
  var colore = false;
  for(i=0; i<td.length; i++){
    if (colore)
      td[i].setAttribute("bgcolor", "yellow");
    colore = !colore;
  }
</script>

</body>
</html>
```

ESEMPIO

Ecco come creare una pagina che, al click su un pulsante, crea una copia del `<div>` originario. Per farlo, l'esempio nell'ordine:

- mostra come aggiungere un evento a un elemento recuperato grazie al suo ID;
- implementa una funzione che recupera il primo dei tag `div`, ne crea una copia e, infine, aggiunge tale copia al `div` di partenza:

```
<html>
<script type="text/javascript">
function aggiungi(){
    var div = document.getElementsByTagName("div");
    div[0].appendChild(div[0].cloneNode(true));
```

▶

```
}
</script>
<body>
  <input type="button" id="b" name="b" />
  <div>
  <p>Un esempio di frase</p>
  </div>
<script type="text/javascript">
    document.getElementById("b").addEventListener("click", aggiungi, false);
</script>
</body>
</html>
```

2.1 Il padre di tutto il DOM: *window*

Esiste un oggetto che è il padre 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 del desktop e così via.

OSSERVAZIONE Ci sono altri metodi che appartengono a `window` per aprire finestre figlie, chiamate *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 a dotarsi di strumenti per il loro blocco. Ecco che chi si trova a creare nuove 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 po-pup può portare a logiche difficilmente replicabili su dispositivi dotati di browser semplificati. Per questi, e per altri motivi sempre legati all'usabilità delle applicazioni, tali metodi sono qui accennati ma non illustrati.

Nella TABELLA **5** è mostrata una sintesi delle principali proprietà di `window`. Si noti come alcune proprietà siano usate da Internet Explorer mentre altri browser usano, per lo stesso tipo di informazione, altre proprietà.

TABELLA **5**

Proprietà o metodo	Significato
`status`	Permette di impostare messaggi sulla barra di stato (in basso alla finestra del browser)
`statusbar`	Se impostata a `false`, la sua proprietà `visible`, nasconde la barra di stato; di default è a `true`
`locationbar`	Se impostata a `false`, la sua proprietà `visible` nasconde la barra degli indirizzi; di default è `true`
`close()`	Metodo che chiude la finestra ▶

▶ TABELLA 5

Proprietà o metodo	Significato
`scrollbars`	Se impostata a `false`, la sua proprietà `visible` nasconde eventuali scrollbar (orizzontale e verticale)
`print()`	Avvia il processo di stampa della pagina (l'utente deve comunque confermare il comando)
`toolbar`	Valore booleano che indica se la toolbar 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, in pixel, della finestra del browser rispetto al desktop (Internet Explorer e altri)
`screenX, screenY`	Rispettivamente, distanza dal bordo sinistro e da quello in alto, in pixel, della finestra del browser rispetto al desktop (Firefox, Chrome e altri)

OSSERVAZIONE Le proprietà `screenX` e `screenY` restituiscono il valore –4 su una finestra massimizzata (su sistemi operativi Windows). Questo perché l'angolo superiore a sinistra della finestra è leggermente al di fuori della porzione visibile dello schermo.

ESEMPIO

Ecco uno script che mostra le coordinate X e Y della finestra del browser rispetto al desktop; si noti come la scelta di restituire una proprietà o l'altra venga preceduta dal test della proprietà stessa; tale test restituisce **`false`** ogniqualvolta la proprietà non esiste (quando una proprietà non esiste, tentare di accedervi dà come risultato `undefined`; tale valore, quando usato come valore di verità, viene valutato a **`false`**):

```
<html>
<script type="text/javascript">
function getX(){
  if (window.screenX)
    return window.screenX;
  else
    return window.screenLeft;
}

function getY(){
  if (window.screenY)
    return window.screenY;
  else
    return window.screenTop;
}
</script>
<body onload="alert('X:'+getX()+', Y:'+getY())">
</body>
</html>
```

L'oggetto `window` espone anche una serie di ulteriori oggetti e collezioni e, in particolare, gli oggetti `navigator`, `location`, `history`, `document` e `screen`. Le caratteristiche principali di questi oggetti saranno ora illustrate e analizzate.

OSSERVAZIONE Essendo `window` l'oggetto principale da cui discendono gli altri, è possibile ometterlo quando si referenzia un qualsiasi oggetto specifico; ecco perché è possibile referenziare l'oggetto `document` con una notazione come `document.write` anziché scrivere, come sarebbe corretto, `window.document.write`. Benché quest'ultima forma sia più prolissa, è equivalente all'altra e può essere usata per rendere esplicito il riferimento a `window`, altrimenti sottinteso.

navigator

L'oggetto `navigator` permette di accedere alle informazioni sul browser che sta eseguendo il codice. Queste informazioni sono alla base delle diverse statistiche che alcuni siti collezionano in merito al tipo e versione di browser, sistema operativo in uso e così via. Nella TABELLA **6** viene illustrato il significato delle principali proprietà, mentre nelle FIGURE **3** e **4** sono mostrate delle immagini ottenute usando browser diversi (Chrome e Firefox, nello specifico) dove il risultato è quello restituito dalla funzione `mostra` a cui viene passato la stringa `'navigator'`.

TABELLA **6**

Proprietà o metodo	Significato
`appCodeName`	Nome in codice del browser (per compatibilità è sempre Mozilla, sia per Firefox sia per Internet Explorer o Chrome)
`appName`	Nome del browser in forma semplice
`appVersion`	Numero di versione; è un valore dal significato «interno», ovvero che può non corrispondere alla versione di rilascio
`userAgent`	Stringa che racchiude le principali informazioni del browser
`platform`	Tipologia di piattaforma software dove viene eseguito il browser
`onLine`	`True` se si è connessi in rete, `false` altrimenti

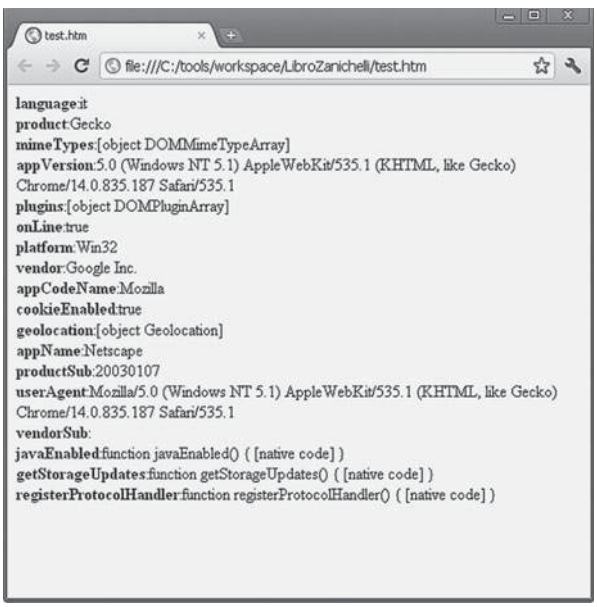


FIGURA **3**

FIGURA **4**

OSSERVAZIONE È comune tentare di risolvere le incompatibilità tra browser diversi cercando di riconoscere il browser in uso, ricorrendo, per esempio, allo `userAgent` e seguendo strade diverse nei casi in cui si riconosca un browser con problemi a interpretare certe proprietà o certi metodi. Purtroppo questa strada non sempre porta a risultati corretti, perché si dovrebbero censire tutti i possibili browser e bisognerebbe mantenere aggiornato tale elenco, cosa alquanto dispendiosa in termini di tempo. Non solo: esistono plug-in per far sì che lo `userAgent` inviato dal proprio browser sia modificato. jQuery, che vedremo a breve, adotta una strategia diversa (comune a molte altre librerie) per evitare un censimento esaustivo: prima di usare una certa proprietà esegue un test con un `if`, proprio come fatto nell'esempio precedente a proposito delle proprietà `screenX` o `screenLeft`.

location

L'oggetto `location` modella l'indirizzo del documento visualizzato nella finestra. In particolare la proprietà `href` è la proprietà che contiene l'URL. Altre proprietà e metodi sono mostrati nella TABELLA **7**.

TABELLA **7**

Proprietà o metodo	Significato
`protocol`	Protocollo in uso (HTTP o HTTPS)
`host`	Nome che identifica l'host
`pathname`	Il percorso dell'attuale pagina
`search`	Contiene la query `string`, ovvero la (eventuale) parte di indirizzo che segue il carattere «?» e specifica dei valori per alcune proprietà
`reload()`	Ricarica la pagina
`replace(url)`	Sostituisce la pagina attuale con quella recuperata dalla URL specificata

OSSERVAZIONE Assegnare un nuovo valore alla proprietà `href` fa sì che si carichi la nuova pagina al posto di quella attuale. Se da tale pagina l'utente fa click sul pulsante «indietro» («back») del browser, viene mostrata la pagina di partenza. Se si usa il metodo `replace`, l'URL viene usata per caricare un nuovo documento, ma l'indirizzo del nuovo sovrascrive il precedente. In questo modo il pulsante «indietro» del browser non riporterà il documento originario ma, se c'è, riporterà quello ancora precedente.

ESEMPIO

Supponendo esista una pagina chiamata test2.htm, questo codice HTML fa sì che, se si preme sul primo pulsante, la pagina test2.htm venga visualizzata; ma è possibile ritornare alla pagina di partenza con il tasto «indietro» del browser. Se invece si preme sul secondo pulsante, test2.htm prende il posto della pagina corrente, ma non v'è traccia della sua visualizzazione, tanto che il pulsante «indietro» risulta disabilitato. ▶

```
<html>
<body>
  <input type="button" value="Nuova"
    onclick="location.href='test2.htm'" />
  <input type="button" value="Sovrascrivi"
    onclick="location.replace('test2.htm')" />

</body>
</html>
```

history

Caricare nuove pagine, come si è visto, permette di seguire un certo «percorso». Ma a livello di applicazione JavaScript è possibile navigare attraverso la storia delle pagine visitate precedentemente rispetto alla pagina dove viene eseguito lo script. Questo avviene grazie all'oggetto `history` e ai metodi che espone. Tali metodi sono mostrati nella TABELLA **8**.

TABELLA **8**

Proprietà o metodo	Significato
`back()`	Simula il comportamento del tasto «back» del browser: porta alla pagina precedente della cronologia
`forward()`	Simula il comportamento del tasto «forward» del browser: porta alla pagina successiva della cronologia
`go(n)`	Va, rispetto alla posizione 0 della pagina corrente, alla pagina specificata. Se il numero passato è negativo, va alla pagina che precede nella cronologia (per esempio –2 va indietro di due pagine); se il numero passato è positivo, va avanti per il numero di volte specificato

ESEMPIO

Ecco come realizzare una pagina HTML che, grazie all'oggetto `history`, permette di andare avanti e indietro di una pagina rispetto alle pagine visitate. L'andare avanti o indietro avviene premendo su uno dei primi due pulsanti. C'è anche la possibilità di «saltare» alla pagina visitata in una posizione specificata da un campo di testo, premendo sul terzo e ultimo pulsante.

```
<html>
<body>
  <input type="button" value="vai indietro" onclick="history.back()"/>
  <input type="button" value="vai avanti" onclick="history.forward()"/>
  <br/>
  Pagina<input type="text" id="num" name="num"/>
  <input type="button" value="vai"
    onclick="history.go(document.getElementById('num').value)"/>
</body>
</html>
```

OSSERVAZIONE I metodi descritti permettono una navigazione basandosi sulla storia delle pagine visitate. Ma si osservi che nessuno di tali metodi permette di recuperare il contenuto o l'indirizzo delle pagine visitate. Questa è una scelta progettuale ben precisa, per evitare che un'applicazione web possa conoscere cosa si è visitato prima di accedere a essa. Si supponga cosa accaderebbe se visitando un sito commerciale, per esempio, esso potesse aver accesso alle pagine già visitate, che potrebbero essere quelle della concorrenza, ma anche quelle della propria banca o di un servizio «riservato». La conoscenza di questi dettagli comporterebbe un notevole rischio per la privacy. In questo modo la privacy è garantita, almeno da questo punto di vista.

screen

L'oggetto `screen` fornisce informazioni sul display dell'utente e sulle sue capacità, in termini di risoluzione video e di colori supportati. Per esempio le proprietà `width` e `height` si riferiscono, rispettivamente, alla risoluzione video in orizzontale e in verticale. Nella FIGURA **5** è mostrato il risultato dell'esecuzione della funzione `mostra`, introdotta in precedenza, e applicata all'oggetto `screen`.

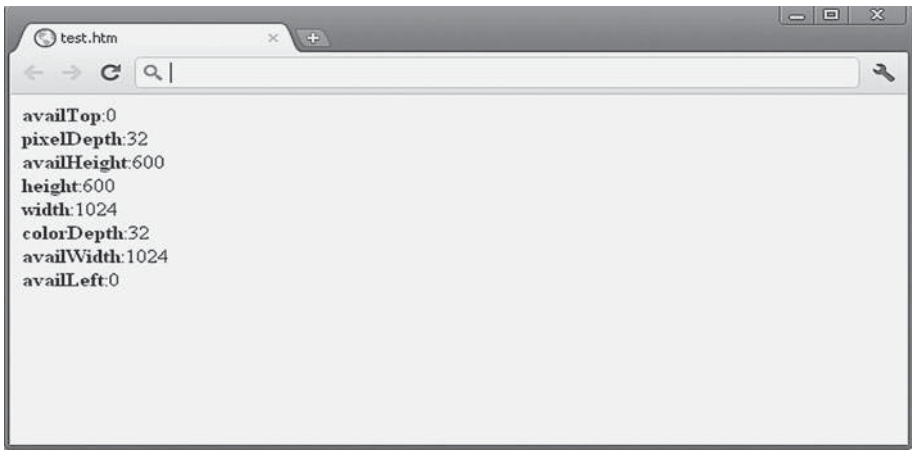


FIGURA **5**

2.2 Collezioni di oggetti di *document*

Il browser espone, come proprietà dell'oggetto `document`, collezioni grazie alle quali è possibile accedere a determinati tipi di elementi di una pagina HTML; in particolare espone le collezioni `anchors` e `links`, che contengono, rispettivamente, tutte le ancore e i link della pagina; `forms`, che contiene tutte le form; `images` per le immagini e `applets` per le Applet Java.
Ciascuna collezione è accessibile sia per nome dell'elemento (se presente) sia per posizione (i numeri partono da 0 per il primo elemento di quel tipo dentro il documento, 1 per il secondo e così via).

ESEMPIO

Partendo da una pagina web è possibile utilizzare il seguente script per generare, in fondo alla pagina, una sequenza di tutte le immagini superiori ai 50 pixel (per evitare di includere icone o altri elementi grafici per la paginazione) usate nel documento. Per comodità le immagini sono visualizzate con una grandezza massima (in orizzontale o in verticale) che è indicata nel parametro della funzione (nell'esempio 100 pixel):

```
<html>
<head>
<script type="text/javascript">
function aggiungiImmagini(dimensione) {
  var len = document.images.length;
  for(i=0; i<len; i++){
    var attuale = document.images[i];
    var max = attuale.width; // dimensione max delle due dimensioni
    if (max<attuale.height){
      max = attuale.height;
    }
    if (max>50) { // va bene, non è un'icona
      var img = document.createElement("img");
      img.setAttribute("src", attuale.src);
      img.setAttribute("width", attuale.width*dimensione/max);
      img.setAttribute("height", attuale.height*dimensione/max);
      // aggiunge l'immagine in fondo al documento
      document.body.appendChild(img);
    }
  }
}
</script>
</head>
<body onload="aggiungiImmagini(100)">
  <!-- contenuto -->
</body>
</html>
```

In FIGURA **6** è mostrato l'esempio applicato alla voce «Italia» di Wikipedia (per generarlo è stato copiato il contenuto della pagina dentro il body dell'esempio precedente e sono stati modificati i riferimenti alle immagini affinché fossero assoluti e non relativi, facendoli riferire al dominio http://it.wikipedia.org).

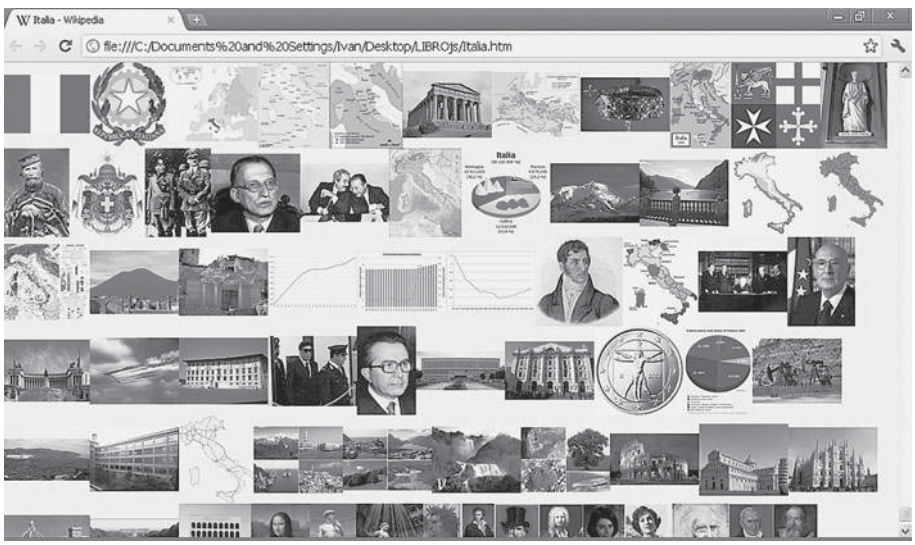


FIGURA **6**

2.3 Accedere agli elementi di una form

Una form può contenere vari elementi come aree di testo, caselle di testo, caselle a scelta (singola o multipla), caselle di selezione (selezione singola, chiamate *radio buttons*, o multipla, chiamate *check boxes*). Ciascuno di essi offre proprietà per leggere e settare valori inseriti e/o selezionati.

ESEMPIO

Per leggere o scrivere il valore di un campo di tipo *text* o *hidden* si può far riferimento alla sua proprietà *value*; ecco una pagina in cui, quando l'utente fa click sul pulsante, il contenuto del campo nascosto viene copiato in quello visibile; solo a titolo di esempio il campo nascosto viene acceduto usando il suo nome (nascosto) come proprietà della form recuperata specificando il suo nome come indice per l'insieme *forms*: `document.forms['nomeForm'].nascosto`. L'altro elemento viene recuperato in maniera simile, ma accedendo alla posizione della form (che è la prima, cioè quella con indice zero): `document.forms[0].visibile.value`.

```
<html>
<head>
<script type="text/javascript">
  function porta(){
    var val = document.forms['nomeForm'].nascosto.value;
    document.forms[0].visibile.value = val;
    return true;
  }
</script>
</head>
<body>
  <form name="nomeForm" action="post">
    <input type="text" name="visibile" value="" />
    <input type="hidden" name="nascosto" value="Una prova" />
    <input type="button" value="Porta" onclick="porta()" />
  </form>
</body>
</html>
```

Nel caso di liste, realizzate grazie al tag `<select>`, le opzioni possono essere un numero qualsiasi; se è ammessa la selezione multipla, anche gli elementi selezionati possono essere più d'uno. A livello di DOM, una volta acceduto all'elemento `select` (grazie al suo ID o al suo nome), ci si può riferire alle sue opzioni grazie all'array `options`. Ciascun elemento dell'array ha una proprietà, `selected`, che permette di sapere se l'elemento è selezionato oppure no.

ESEMPIO

La seguente pagina mostra una form con un campo di inserimento testo e due pulsanti. Facendo click sul primo pulsante il contenuto del campo viene aggiunto a una lista; premendo sul secondo pulsante vengono mostrati i valori selezionati:

```
<html>
<head>
<script type="text/javascript">
  function aggiungi(){
```

▶

```
    var val = document.forms['nomeForm'].visibile.value;
    var opt = document.forms['nomeForm'].lista.options;
    opt.length = opt.length +1;
    opt[opt.length-1] = new Option(val, val);
  }
  function mostra(){
    var val = "";
    var opt = document.forms['nomeForm'].lista.options;
    for(i=0; i<opt.length; i++)
      if (opt[i].selected)
        val = val + opt[i].value + "; ";
    if(val=="")
      alert("Nessun elemento selezionato");
    else
      alert("Valori selezionati: "+val);
  }
</script>
</head>
<body>
  <form name="nomeForm" action="post">
  <input type="text" name="visibile" value="" />
  <input type="button" value="Aggiungi" onclick="aggiungi()" />
  <input type="button" value="Mostra" onclick="mostra()" />
  <select name="lista" multiple="multiple">
    <option value="">Selezionare voci…
      </select>
  </form>
</body>
</html>
```

`Option` è un costruttore per l'oggetto che rappresenta un'opzione e possiede due parametri: il valore e la descrizione dell'opzione da creare.

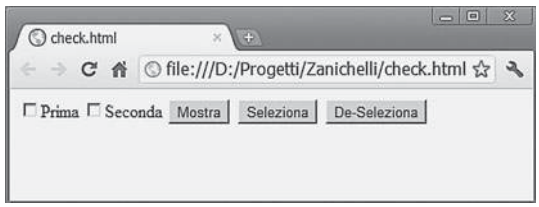
Per quanto riguarda le check list, anch'esse sono gestite attraverso un vettore di elementi; la proprietà che indica se gli elementi sono selezionati o meno è `checked`. Tale proprietà è anche in scrittura; se la si assegna a **`true`** l'elemento viene spuntato; se a **`false`** non viene spuntato.

ESEMPIO

Quando l'utente fa click sul primo pulsante, vengono letti e mostrati i valori spuntati da una checklist; altri due pulsanti permettono, rispettivamente, di selezionare e deselezionare tutti gli elementi.

```
<html>
<head>
<script type="text/javascript">
  function assegna(cosa){
    var ck = document.forms['nomeForm'].test;
    for (i = 0; i < ck.length; i++)
      ck[i].checked = cosa;
  }
```

```
function selezionati(){
  var val = "";
  var opt = document.forms['nomeForm'].test;
  for(i=0; i<opt.length; i++)
    if (opt[i].checked)
      val = val + opt[i].value + "; ";
  if(val=="")
    alert("Nessun elemento selezionato");
  else
    alert("Valori selezionati: "+val);
  }
</script>
</head>
<body>
  <form name="nomeForm" action="post">
    <input type="checkbox" name="test" value="1">Prima
    <input type="checkbox" name="test" value="2">Seconda
    <input type="button" value="Mostra" onclick="selezionati()" />
    <input type="button" value="Seleziona" onclick="assegna(true)" />
    <input type="button" value="De-Seleziona"
onclick="assegna(false)" />
  </form>
</body>
</html>
```



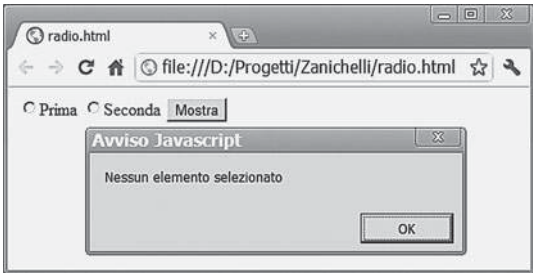
Quando, in una lista di elementi, se ne può selezionare solo uno e il numero degli elementi non è elevato, i *radio buttons* possono essere considerati una buona scelta.

ESEMPIO

Ecco una pagina che permette di selezionare una sola opzione grazie ai *radio buttons*; se l'utente preme sul pulsante viene mostrato il valore dell'elemento eventualmente selezionato:

```
<html>
<head>
<script type="text/javascript">
function selezionati(){
    var val = "";
    var opt = document.forms['nomeForm'].test;
    for(i=0; i<opt.length; i++)
      if (opt[i].checked)
        val = val + opt[i].value + "; ";
    if(val=="")
```

```
        alert("Nessun elemento selezionato");
      else
        alert("Valori selezionati: "+val);
    }
</script>
</head>
<body>
  <form name="nomeForm" action="post">
    <input type="radio" name="test" value="1">Prima
    <input type="radio" name="test" value="2">Seconda
    <input type="button" value="Mostra" onclick="selezionati()" />
  </form>
</body>
</html>
```



2.4 Accedere e modificare i CSS

Attraverso l'utilizzo del DOM è possibile anche specificare le proprietà normalmente gestite via CSS. Infatti è possibile accedere alla proprietà `style`, come a qualsiasi altra proprietà del DOM. Successivamente si può accedere alle specifiche proprietà CSS come se fossero le proprietà di stile scritte dentro `style`.

ESEMPIO

Di seguito un esempio di pagina HTML con tanti `<div>`. Da una lista di selezione è possibile decidere quale valore dare alla proprietà *display* di tutti i `div`; i `div` sono recuperati grazie al metodo `getElementsByTagName`.

```
<html>
<head>
<script type="text/javascript">
function visualizza(come){
  if(come){
    var divs = document.getElementsByTagName("div");
    for (i = 0; i < divs.length; i++)
      divs[i].style.display = come;
  }
}
</script>
</head>
```

```
<body>
  <select onchange="visualizza(this.value)">
    <option selected="selected">seleziona…</option>
    <option value="none">none</option>
    <option value="inline">inline</option>
    <option value="block">block</option>
  </select>
  <div>Un primo div di esempio</div>
  <div>Un secondo div di esempio</div>
  <div>Un terzo div di esempio</div>
  <div>Un quarto div di esempio</div>
  <div>Un quinto div di esempio</div>
</body>
</html>
```

OSSERVAZIONE Come si è mostrato nell'esempio, la proprietà CSS `display` ha un corrispettivo uguale come proprietà JavaScript della proprietà `style`. È per questo che si è potuto scrivere `style.display`. Questo è vero per tutte le proprietà dai nomi semplici. I nomi composti usano una convenzione leggermente diversa: eventuali nomi di proprietà CSS contenenti trattini hanno il corrispettivo JavaScript senza trattini ma con l'iniziale maiuscola (che non vale per la prima lettera che è sempre minuscola). Per esempio la proprietà CSS `font-weight` diventa la proprietà JavaScript `style.fontWeight`.

ESEMPIO

```
<html>
<head>
<script type="text/javascript">
function bold(cosa){
  cosa.style.fontWeight = 'bold';
}
</script>
</head>
<body onload="bold(document.getElementsByTagName('p')[0])">
  <p>Un esempio di testo</p>
  <p>Un esempio di testo</p>
</body>
</html>
```

OSSERVAZIONE La proprietà `style` è stata introdotta da Microsoft e resa standard solo con il *Document Object Model (DOM) Level 2 Style Specification.*

Benché, a livello teorico, sia possibile impostare tutti gli attributi di stile desiderati via JavaScript, è buona regola non farne un uso estensivo o, meglio ancora, evitarlo del tutto. Come per la scrittura di pagina HTML si prefe-

risce non inserire gli stili come attributi degli elementi, ma si assegnano a essi classi logiche e poi, a livello di CSS, si definiscono le caratteristiche di formato della classe, così quando si scrive codice JavaScript è preferibile limitarsi alla manipolazione del DOM e assegnare ai vari elementi la classe di stile di loro competenza, demandando a un foglio di stile CSS la sua definizione. In questo caso si ricorre alla proprietà `className` dell'elemento.

ESEMPIO

```
<html>
<head>
<style>
   .grassetto{font-weight:bold}
   .normale{font-weight:normal}
</style>

</head>
<body
   onload="document.getElementsByTagName('p')[0].className='grassetto'">
   <p class="normale">Un esempio di testo</p>
   <p class="normale">Un esempio di testo</p>
</body>
</html>
```

In pratica assegnare un nuovo valore alla proprietà `className` equivale a sostituire l'eventuale classe esistente o a crearne una nuova. Se la si vuole eliminare allora si può assegnare la stringa vuota; se invece si vuol aggiungere una nuova classe a eventuali classi esistenti basta concatenare il nome della nuova classe (lasciando uno spazio tra le due).

ESEMPIO

Ecco una pagina che fa uso della funzione `aggiungi`, che imposta i nomi delle classi a grassetto, sottolineato e italico (tutte insieme, una di seguito all'altra):

```
<html>
<head>
<style>
   .grassetto{font-weight:bold}
   .normale{font-weight:normal}
   .sottolineato{text-decoration:underline}
   .italico{font-style:italic}
</style>
<script type="text/javascript">
```

▶

```
function aggiungi(cosa){
  cosa.className = "grassetto";
  cosa.className += "sottolineato";
  cosa.className += "italico";
}
</script>
</head>
<body onload="aggiungi(document.getElementsByTagName('p')[0])">
  <p class="normale">Un esempio di testo</p>
  <p class="normale">Un esempio di testo</p>
</body>
</html>
```

OSSERVAZIONE Se due classi definiscono una stessa proprietà, è importante l'ordine con cui esse sono state definite. Ha precedenza la classe che viene prima. Nell'ultimo esempio la classe `'grassetto'` non viene aggiunta ma assegnata. Questo perché, se la si aggiunge alla classe normale, entrambe definiscono la proprietà `font-weight`, con il risultato che l'aggiunta di `'grassetto'` non avrebbe effetto.

2.5 Come creare applicazioni complesse

La realizzazione di applicazioni complesse comporta l'utilizzo di numerosi oggetti o funzioni. Anziché copiare parti di codice simili in tanti file HTML 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 anziché ricaricarli; 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), presentazione (CSS) e comportamento (JavaScript) agevola la comprensione delle singole parti e semplifica il processo di sviluppo e offre la possibilità di una suddivisione di compiti all'interno dei gruppi di lavoro;
- una netta separazione semplifica anche la validazione delle pagine (i file HTML e i CSS possono essere validati separatamente, mentre il codice JavaScript non va a «sporcare» la pagina originaria con i caratteri usati).

Accanto a questa raccomandazione è bene valutare sempre 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, 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 degli script non è possibile.
Detto questo, rimane un grosso problema da affrontare e risolvere: la scrittura di programmi che possono essere eseguiti su browser diversi; un modo per semplificare questo problema è rappresentato dalle librerie che, anche per i metodi base come l'accesso al DOM e alla sua gestione, forniscono API di più alto livello e si fanno carico delle eventuali incompatibilità a basso livello. Di seguito verrà analizzata una di queste librerie: **jQuery**.

3 Il ruolo delle librerie

Browser diversi possono presentare incompatibilità a vari livelli; per esempio alcune versioni di browser forniscono supporto parziale alle specifiche del DOM W3C, altre offrono implementazioni errate, altre ancora forniscono modi alternativi per fare certe operazioni. Scrivere codice complesso che tratti tutte queste differenze richiede una conoscenza specialistica dei diversi browser, nonché notevole disciplina per gestire correttamente le differenze senza scrivere codice poco mantenibile.
Appresi i fondamenti del linguaggio e il suo utilizzo dentro le pagine web, è possibile sviluppare applicazioni più o meno complesse. Man mano che aumenta la complessità ci si accorge però che JavaScript possiede alcuni svantaggi: l'interazione con il DOM è soggetta a specificità legate al browser che esegue la pagina, la sintassi del linguaggio a volte è prolissa e può portare a errori; ci vuole molta organizzazione per mantenere il codice correttamente strutturato.

ESEMPIO

Come già accennato, browser differenti possono avere diversità di comportamenti sulla stessa pagina. Uno degli esempi più eclatanti di difformità di comportamento è rappresentato dall'implementazione del metodo `getElementById`. La sua semantica dovrebbe essere, come dice il nome stesso, di recuperare l'elemento della pagina il cui `id` è passato come argomento. Vecchie versioni di Internet Explorer come la 6 e la 7, per esempio, implementavano questo metodo con un comportamento strano: per certi elementi cercava l'attributo `name` anziché l'attributo `id`. Questo fatto può essere verificato molto semplicemente usando questa pagina di prova:

```
<html>
<head>
<script type="text/javascript">
  function mostra(id) {
    var v = document.getElementById(id);
    alert(v.innerHTML);
  }
</script>
</head>
```

```
<body>
  <form name="ID2" id="ID1" action="">
    <div>
      <h2>FORM con name=ID2, id=ID1</h2>
    </div>
  </form>
  <div id="ID2">
    <h2>DIV con ID2</h2>
  </div>
  <input type="button" onclick="mostra('ID1')" value="id='ID1'" />
  <input type="button" onclick="mostra('ID2')" value="if='ID2'" />
</body>
</html>
```

Premendo sul primo pulsante ci si aspetta la visualizzazione del contenuto della form, premendo sul secondo ci si aspetta il contenuto del secondo *div*.
Nelle FIGURE **7** e **8** è mostrato il comportamento, corretto, di Chrome e quello errato di Explorer: quest'ultimo, premendo un pulsante qualsiasi, recupera sempre il contenuto della form.

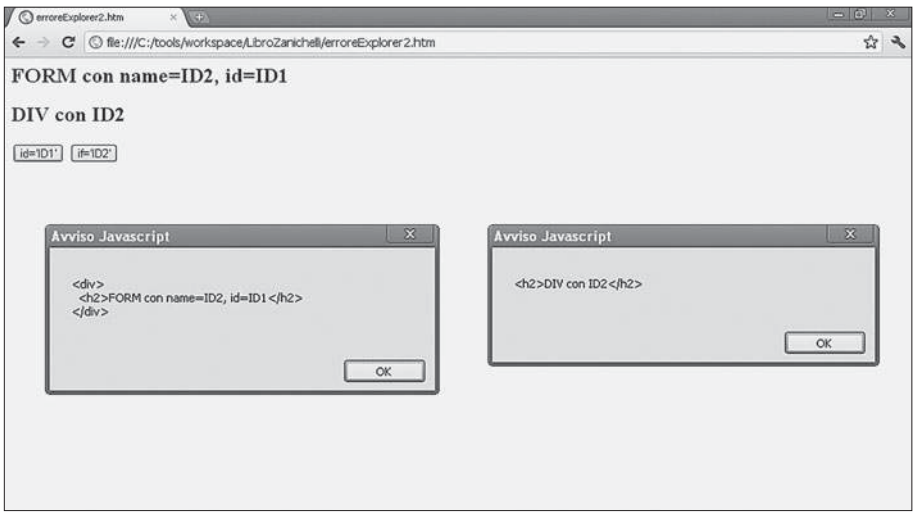


FIGURA **7**

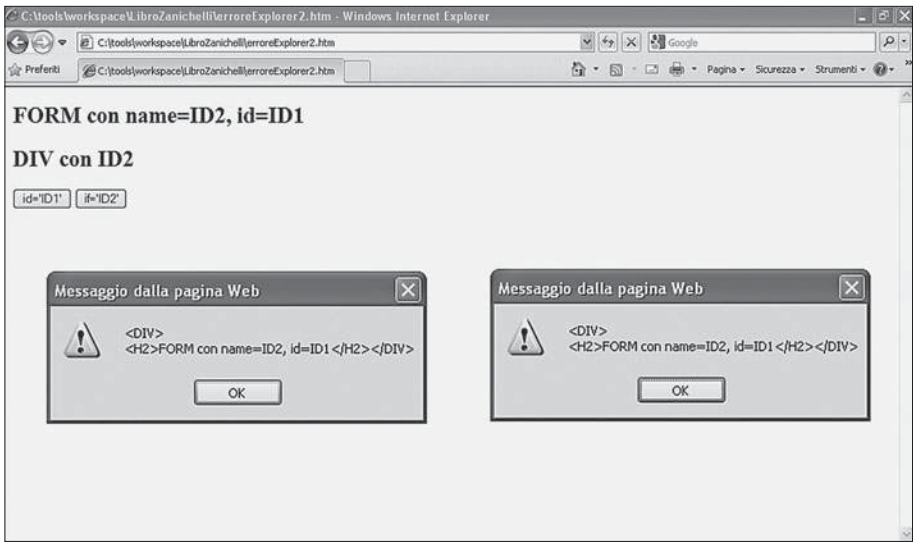


FIGURA **8**

Esistono anche operazioni di per sé complesse da gestire; funzionalità come il *drag&drop* di oggetti visuali o altri effetti avanzati necessitano di molto codice. Tutto questo può essere semplificato grazie a una libreria JavaScript che si prenda carico di vari aspetti, permettendo di concentrarsi sul solo risultato desiderato.
Esistono numerose librerie: jQuery, Prototype, MooTools, YUI Library, Dojo Toolkit sono tutte librerie diffuse, ben documentate e supportate da un'ampia schiera di sviluppatori. In questo contesto si analizzerà jQuery, una delle librerie più utilizzate.

3.1 jQuery

jQuery è una libreria JavaScript scaricabile da Internet che si pone come obiettivo principale quello di semplificare la manipolazione e la gestione di documenti HTML. Lo fa in maniera *cross browser*, con del codice fortemente ottimizzato (sia in velocità che come «pesantezza», ovvero quantità di caratteri da scaricare per includere la libreria).

ESEMPIO

Si scarichi l'ultima versione della libreria jQuery e, salvatala in locale o sul server dove ci sono le pagine HTML, la si includa usando il tag mostrato di seguito dentro l'header della pagina:

```
<head>
  <!-- . . . -->
    <script src="jquery.js"></script>
  <!-- . . . -->
</head>
```

Avere lo script in locale può avere il vantaggio di essere certi di avere disponibile la versione desiderata.

ESEMPIO

È una pratica diffusa quella di utilizzare URL assolute, come, per esempio:

```
<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js"></script>
```

Tale URL è piuttosto complessa; la prima parte è standard e specifica che si desidera caricare la libreria jQuery dal repository ufficiale. Poi si specifica la versione; nell'esempio è la 1.6.2. Infine il nome del file ha, in questo caso, la parola «.min». Indica la versione minimizzata della libreria; in pratica sono stati tolti tutti gli spazi inutili e sono stati usati nomi di variabili e funzioni il più corti possibile. Tutto questo a scapito della leggibilità ma a guadagno del numero di byte da scaricare. Una simile libreria non si presta certo all'analisi del suo codice; se lo si vuol fare, si tolga il «.min» e si avrà a disposizione la libreria in formato «leggibile». Togliendo «.min» si ha la versione leggibile, indispensabile per indagare i contenuti della libreria.

OSSERVAZIONE Usare una URL assoluta ha il vantaggio che, se l'utente ha già visitato una pagina web che faceva uso della stessa versione di jQuery, il browser ce l'ha in cache ed evita di eseguire un nuovo down-

load. Ma anche se non dovesse averla in cache, è probabile che il download sia più veloce da un server fortemente ottimizzato piuttosto che da gran parte dei server dove potrebbe stare la pagina HTML che fa uso della libreria.

Selettori

Una delle operazioni di base è quella di selezionare un elemento della pagina; essendo un'operazione molto comune, jQuery ha messo a disposizione una funzione, chiamata *selettore*, che ha la forma più breve possibile utilizzando un solo carattere: `$`. Esso si applica in questo modo:

```
$(nome)
```

Quando il nome è preceduto dal simbolo # allora cerca l'elemento HTML con quell'ID.

ESEMPIO

Tramite l'utilizzo di jQuery, anche versioni di browser in cui il `getElementById` poteva fallire ora funzionano correttamente usando la notazione `$("#id")`. L'esempio visto in precedenza diventa pertanto:

```
<html>
<head>
<script src="jquery.js"></script>
<script type="text/javascript">
  function mostra(id) {
    var v = $("#"+id);
    alert(v.html());
  }
</script>
</head>
<body>
  <form name="ID2" id="ID1" action="">
    <div>
      <h2>FORM con name=ID2, id=ID1</h2>
    </div>
  </form>
  <div id="ID2">
    <h2>DIV con ID2</h2>
  </div>
  <input type="button" onclick="mostra('ID1')" value="id='ID1'" />
  <input type="button" onclick="mostra('ID2')" value="if='ID2'" />
</body>
</html>
```

Chiaramente il problema adesso non si pone: è jQuery a farsene carico e a risolverlo. Per chi è interessato a capire come, basta guardare il sorgente. Si apra la pagina e si inizi a scorrere il sorgente. A un certo punto compare questo commento:

```
// HANDLE: $("#id").
```

Siamo giunti al punto che ci interessa; quello che segue è: ▶

```
elem = document.getElementById(match[2]);
//Check parentNode to catch when Blackberry 4.6 returns
//nodes that are no longer in the document #6963
if (elem && elem.parentNode) {
// Handle the case where IE and Opera return items
// by name instead of ID
  if (elem.id !== match[2]) {
    return rootjQuery.find(selector);
  }
// Otherwise, we inject the element directly into the jQuery object
  this.length = 1;
  this[0] = elem;
}
```

Come si può osservare dal commento interno, c'è un test specifico proprio per trattare correttamente il caso di IE (e, leggendo il commento, si viene a sapere che questo caso particolare può avvenire anche con Opera, un altro web browser).
L'analisi completa dei sorgenti di jQuery non è immediata; questa semplice ricerca ci aiuta a capire come l'uso di una libreria astrae dai dettagli e dalle incompatibilità fornendo primitivi di alto livello, sicuri e testati da numerosi sviluppatori (ovviamente errori e bug sono sempre possibili anche in jQuery, come in qualsiasi programma software).

OSSERVAZIONE Il nome stesso della libreria, jQuery, ricorda che una delle sue caratteristiche fondamentali è l'interrogazione sugli elementi del DOM e una gestione il più possibile omogenea e semplificata degli stessi. Si può osservare che il modo di riferirsi a un elemento con uno specifico id, #nomeId, è lo stesso che si usa a livello di selettori CSS. Questa non è assolutamente una casualità ma una scelta progettuale ben precisa di jQuery: essa riusa tutti i selettori CSS per selezionare gli elementi del DOM. Di seguito i selettori CSS:

```
<style>
p{
  color : red; //imposta a rosso tutti i paragrafi, tag <p>
}

.nomeC{
  color : red; //imposta a rosso tutti i tag la cui classe è nomeC
}

#qualeId{
  color : red; //imposta a rosso l'elemento il cui id è qualeId
}
</style>
```

e, analogamente, un corrispettivo esempio d'uso dei selettori su jQuery:

```
$("p").css('color', 'red');
$(".nomeC").css('color', 'red');
$("#qualeId").css('color', 'red');
```

A differenza dei CSS, che permettono di specificare solo uno specifico stile di visualizzazione, una volta definiti gli elementi a cui applicare lo stile, jQuery permette, sugli elementi selezionati, di modificare lo stile, il contenuto (compresa l'aggiunta o rimozione di figli) ma anche il comportamento, dove per comportamento si intende la possibilità di rispondere e gestire gli eventi.

jQuery è molto flessibile e versatile e permette di usare una gran varietà di selettori. Nella TABELLA **9** sono dati alcuni esempi.

TABELLA **9**

Selettore	Significato
`$("p")`	Recupera gli elementi il cui tag è `p` (ovvero sono dei paragrafi)
`$("#qualeId")`	Recupera l'elemento con id `'qualeId'`
`$(".qualeC")`	Recupera gli elementi la cui classe è `qualeC`
`$("p.qualeC")`	Recupera tra tutti gli elementi il cui tag è `p` solo quelli la cui classe è `qualeC`
`$("span p")`	Recupera gli elementi il cui tag è `p` e sono all'interno di un tag `span`
`$("form[id^=a]")`	Tutte le form i cui id iniziano per `'a'`
`$("a[target=_blank]")`	Tutti i link il cui target è `_blank`
`$("tr:odd")`	Tutte le righe dispari di una tabella
`$("li:first")`	Primo elemento di una lista

OSSERVAZIONE L'importanza dei selettori è presto detta: prima di operare su un qualsiasi elemento, esso deve essere selezionato. Poi, usando la «catena» dei metodi, si possono applicare le azioni desiderate. Per «catena» si intende la possibilità di usare dei metodi in cascata ad altri; per esempio:

```
$("p").addClass("qualeClasse").show()
```

3.2 Script eseguiti al caricamento della pagina

Gli script veri e propri si possono inserire, come si è visto, in qualunque parte della pagina. Talvolta è desiderabile che alcune funzioni vengano eseguite subito, non appena la pagina viene caricata. C'è però un inconveniente: se i sorgenti modificano e/o interagiscono con elementi della pagina, bisogna essere sicuri che, nel momento in cui lo script viene eseguito, tali elementi siano a disposizione per essere manipolati.

ESEMPIO

jQuery mette a disposizione un'utile funzione che permette di eseguire il codice solo quando tutto il documento è stato caricato:

```
$(document).ready(function(){
    // codice da eseguire
});
```

In pratica si seleziona il documento della pagina e si dice che quando è pronta (*ready*, ovvero caricata) si può mandare in esecuzione il codice contenuto nella funzione racchiusa.
Si supponga di avere una pagina HTML con una *textarea* e due pulsanti.

ESEMPIO

```
<textarea id="tabella">
jQuery(selector).animate(proprietà, [durata], [easing], [callback])
</textarea>
  <input type="button" id="pulsante-piu" value="+" />
  <input type="button" id="pulsante-meno" value="-" />
```

Si vorrebbe che, facendo clic sul primo pulsante, la *textarea* venisse ingrandita, mentre venisse rimpicciolita facendo clic sul secondo. Per farlo si farà uso della funzione `animate`. Essa ha come unico argomento obbligatorio un insieme di proprietà, cioè una qualsiasi proprietà numerica dei CSS. Ecco come potrebbe essere il codice JavaScript che, usando jQuery, effettua le animazioni volute.

ESEMPIO

```
$(document).ready(function() {
  $("#pulsante-piu").click(function() {
    $("#tabella").animate({
      height: "200px",
      width:  "600px",
    })
  });
$("#pulsante-meno").click(function() {
    $("#tabella").animate({
      height: "20px",
      width:  "60px",
    })
  });
});
```

La pagina di esempio complessiva risulta essere la seguente; nelle FIGURE **9** e **10** sono mostrate le immagini della pagina ottenuta premendo sul pulsante «+» e quella ottenuta dopo aver premuto il pulsante «–».

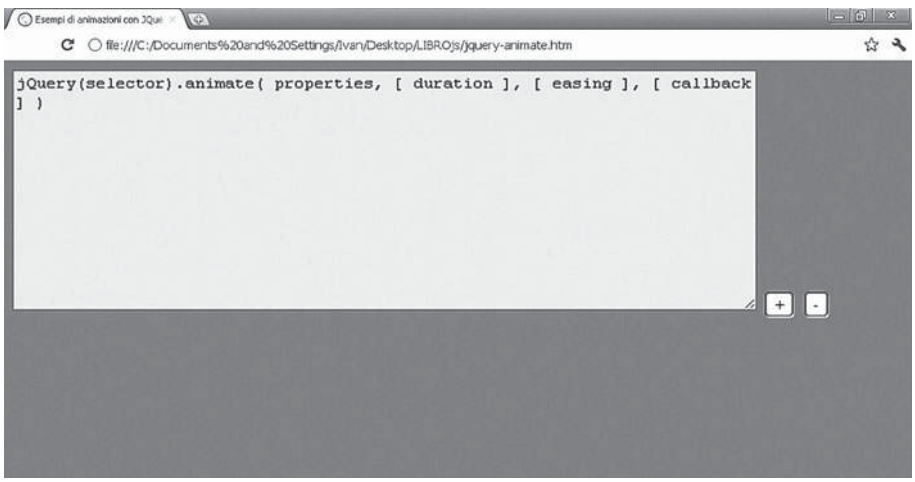


FIGURA **9**

FIGURA **10**

ESEMPIO

```
<html>
<head>
  <script src="jquery.js"></script>
</head>
<body bgcolor="red">
<textarea id="tabella">
jQuery(selector).animate(properties, [duration], [easing], [callback])
</textarea>
  <input type="button" id="pulsante-piu" value="+" />
  <input type="button" id="pulsante-meno" value="-" />
  <script>
  $(document).ready(function() {
    $("#pulsante-piu").click(function() {
      $("#tabella").animate({
        height: "200px",
        width:  "600px",
          opacity: "0.5"
      })
    });
    $("#pulsante-meno").click(function() {
      $("#tabella").animate({
        height: "20px",
        width:  "60px",
          opacity: "1"
      })
    });
});
  </script>
  </body>
</html>
```

OSSERVAZIONE Esistono dei moduli scritti da altre persone, chiamati *plug-in*, che realizzano caratteristiche aggiuntive rispetto a quelle previste dalla libreria jQuery.

3.3 jQuery e l'accesso al DOM

Anche l'accesso e la modifica del DOM vengono realizzati da jQuery attraverso opportune funzioni interoperabili tra diversi browser. In particolare, l'accesso all'HTML di un elemento, grazie alla funzione `html()`; agli attributi di un elemento avviene, sia in lettura sia in scrittura, grazie alla funzione `attr()`; l'accesso alle proprietà CSS grazie alla funzione `css()`; infine si può usare la funzione `val()` per gestire i valori di un campo di una form. Ecco alcuni esempi che ne illustrano caratteristiche e potenzialità.

html()

ESEMPIO

La funzione `html()` è usata per recuperare il contenuto HTML di un elemento:

```
<html>
<head>
<script src="jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
     alert($("div").html());
  });
</script>
</head>
<body>
  <div>
    <b>Un esempio</b> d'uso di <i>JQuery</i>
  </div>
  <div>
    <b>Altro div</b>
  </div>
</body>
</html>
```

ESEMPIO

Anche nel caso in cui il selettore a cui si applica restituisca più di un elemento, `html()` si applica solo al primo elemento recuperato; se si desidera applicare la funzione `html()` a ciascun elemento del risultato, è necessario scorrere gli elementi con la funzione `each()`:

```
<html>
<head>
<script src="jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
     $("div").each(function(){alert($(this).html());});
  });
</script>
</head>
<body>
  <div>
    <b>Un esempio</b> d'uso di <i>JQuery</i>
  </div>
  <div>
    <b>Altro div</b>
  </div>
</body>
</html>
```

OSSERVAZIONE Le funzioni di recupero dei valori, come `html()` ma, come vedremo, anche `css()`, `attr()` e `val()`, si applicano sempre al primo degli elementi di un eventuale insieme di elementi recuperati da un selettore. Resta valida la regola che per recuperare i valori di ciascun elemento si deve iterare sugli elementi utilizzando il costrutto `each()`.

La funzione `html()` può essere usata anche per settare il contenuto degli elementi. In questo caso si deve specificare un parametro che è o una stringa contenente il codice HTML da settare, o una funzione, la quale ha la responsabilità di restituire il codice HTML da usare.

ESEMPIO

A differenza della funzione senza parametri, usata per leggere il contenuto HTML, la funzione di settaggio si applica a tutti gli elementi restituiti dal selettore:

```
<html>
<head>
<script src="jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
       $("div").html(
         function(index, oldHtml){
           return oldHtml+"<p>in coda</p>";});
         }
       );
  </script>
</head>
<body>
     <div>
       <b>Un esempio</b> d'uso di <i>JQuery</i>
     </div>
     <div>
       <b>Altro div</b>
     </div>
</body>
</html>
```

css()

Per ottenere informazioni su una proprietà CSS, si utilizza il metodo `css()` a cui si passa come argomento il nome della proprietà di cui leggere il valore.

ESEMPIO

```
<html>
<head>
<script src="jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
       alert($("b").css('color'));
  });
```

▶

```
</script>
</head>
<body>
     <div>
       <b>Un esempio</b> d'uso di <i>JQuery</i>
     </div>
     <div>
       <b>Altro div</b>
     </div>
</body>
</html>
```

OSSERVAZIONE Il metodo `css()` si fa carico di gestire diverse notazioni per quanto riguarda il nome della proprietà. Infatti è possibile specificare tale nome sia secondo la notazione CSS, che per i nomi composti utilizza un trattino di separazione (è il caso, per esempio, della proprietà `font-weight`), sia specificando la notazione JavaScript per le proprietà, che prevede l'eliminazione del trattino e la lettera maiuscola per la lettera che seguiva il trattino (nell'esempio `fontWeight`). Non solo: il metodo `css()` si fa carico anche di gestire eventuali differenze e incompatibilità tra browser diversi. È il caso della proprietà `float`. Essa viene referenziata da Internet Explorer come `styleFloat`, da altri browser (in conformità allo standard W3C) con `cssFloat`. È indifferente usare come nome della proprietà `'float'`, `'styleFloat'` o `'cssFloat'`. Attenzione: se si specifica un nome con un trattino, tale nome deve essere racchiuso tra apici (singoli o doppi), mentre un nome di proprietà che non ne fa uso non ha questo vincolo.

Anche il metodo `css()` può essere usato per settare delle proprietà. In particolare esistono tre versioni del metodo; due sono quelle «usuali»: la coppia nome proprietà/valore e nome proprietà/funzione che restituisce il valore da settare. Il terzo metodo permette di specificare un oggetto con i nomi delle proprietà e i valori da assegnare.

ESEMPIO

```
<html>
<head>
<script src="jquery.js"></script>

<script type="text/javascript">
$(document).ready(function(){
       $("i").css('color', 'green');
       $("b").css('color', function(){return 'blue'});
       $("div").css({fontWeight:'bold', color:'red'});
  });
</script>

</head>
```

```
<body>
     <div>
       <b>Un esempio</b> d'uso di <i>JQuery</i>
     </div>
     <div>
       <b>Altr o div</b>
     </div>
</body>
</html>
```

attr()

Il metodo `attr()` permette la lettura dei valori di un attributo; l'attributo di interesse viene passato come stringa al metodo.

ESEMPIO

```
<html>
<head>
<script src="jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
       alert($("a").attr('href'));
  });
</script>
</head>
<body>
     <a href="test.html">Link</a>
</body>
</html>
```

In maniera simile a `css()`, anche il metodo `attr()` possiede tre tipi di invocazione per settare i valori degli attributi: coppie nome/valore, nome/funzione o oggetto.

ESEMPIO

```
<html>
<head>
<script src="jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
       $("a").attr('href', 'altro.html');
       $("a").attr('target', function() {return '_blank'});
       $("table").attr({border:1, align:'right'});
  });
</script>
</head>
<body>
     <table>
     <tr><td><a href="test.html">Link</a></td></tr>
     </table>
</body>
</html>
```

val()

Il metodo `val()` può essere usato per recuperare il valore di un elemento di una form; quando l'elemento può assumere al più un valore, restituisce una stringa, quando ne può restituire più d'uno, restituisce un array di stringhe.

ESEMPIO

```
<html>
<head>
<script src="jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
       alert(
          $("#a").val() +
          $("#b").val() +
          $("#c").val() +
          $("#d").val()
       );
  });
</script>
</head>
<body>
     <input type="text" id="a" value="a" />
     <input type="radio" checked="checked" id="b" value="b" />
     <input type="checkbox" checked="checked" id="c" value="c" />
  <select id="d">
     <option value="d" selected="selected">d</option>
  </select>

</body>
</html>
```

OSSERVAZIONE Si noti l'enorme semplificazione nell'utilizzare il metodo `val()` anziché ricordarsi i diversi metodi per accedere ai diversi tipi di elementi di una form.

4 AJAX e le Google *Maps*

4.1 Esecuzione su singolo thread

È importante tener presente che l'esecuzione di uno script blocca il browser fino al termine della sua esecuzione; questo avviene perché il browser non sa se lo script va a modificare i contenuti della pagina e, in questo caso, rendere parallele due o più istruzioni porterebbe a risultati non determinabili a priori. Ecco che se tale script ha un tempo di esecuzione elevato, l'utente ha la sensazione che il programma sia bloccato. Questa situazione è facilmente simulabile utilizzando uno script con un ciclo che non termina.

ESEMPIO

```
<html>
<body>
<script type="text/javascript">
  for(var i=0; i<10; i++)
    i--;
</script>
</body>
</html>
```

Alcuni browser moderni segnalano questa situazione (script la cui esecuzione dura a lungo) come anomala e chiedono all'utente se vuol continuare la sua esecuzione o bloccarla (FIGURA **11**).

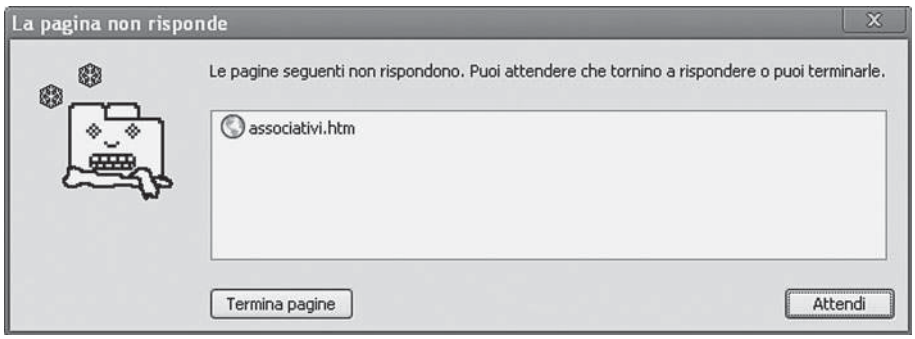


FIGURA **11**

OSSERVAZIONE Anche l'*alert* blocca l'esecuzione degli script: finché l'utente non chiude la finestra il resto del documento non viene caricato. In pratica esiste un unico thread che fa il rendering della pagina e l'esecuzione degli script.

4.2 AJAX e le funzioni asincrone

In certi contesti, in particolare per codice JavaScript con elaborazioni molto lunghe o perché si deve aspettare che una fonte esterna mandi i dati, è necessario non dover attendere la fine dell'esecuzione dello script.
Nel caso di fonte dati esterna esistono oggetti esposti dal browser che permettono di inviare una richiesta a un sito remoto, indicando una funzione che dovrà essere richiamata quando la risposta è pronta; in questo modo il programma può continuare la sua esecuzione senza preoccuparsi dei tempi di risposta.

OSSERVAZIONE Il meccanismo di invocazione asincrona verso fonti di dati esterne rende possibile sviluppare applicazioni che fanno uso del cosiddetto AJAX, *Asynchronous JavaScript And XML* (*JavaScript asincrono e XML*), che prevede una comunicazione dati asincrona e in formato XML per aggiornare i contenuti delle pagine, senza dover cambiare pagina per vedere nuovi dati. Uno degli esempi è rappresentato dalle Google *Maps*, che verranno analizzate nel seguito.

Programmi e thread

Un browser esegue progressivamente il rendering di una pagina HTML permettendo il download simultaneo di eventuali risorse esterne, come le immagini o altri elementi multimediali. C'è da chiedersi come mai non esegue in maniera analoga anche del codice JavaScript.
Il problema è che, se il codice JavaScript contenesse delle istruzioni di creazione di parti di pagina, l'ordine di esecuzione di script diversi renderebbe tale costruzione non deterministica. Pertanto è possibile indicare al browser la possibilità di eseguire porzioni di codice in maniera concorrente indicando in maniera esplicita che tali porzioni non modifichino la pagina che si sta caricando. Tale indicazione avviene specificando l'attributo `deferred` sul tag che include il codice JavaScript.

Nel caso di script JavaScript, dentro la pagina o esterni, la loro esecuzione è, come si è visto, su thread singolo. Questo significa che, se ci sono più file esterni inclusi, la loro inclusione ed esecuzione deve avvenire in sequenza, uno dopo l'altro. Se alcuni script hanno calcoli complessi, essi bloccano tutti gli altri script e il caricamento della pagina.

ESEMPIO

Si supponga di avere tre file, chiamati *file.js*, *file2.js* e *file3.js*, la cui esecuzione dura alcuni secondi; il loro contenuto è uguale a meno dell'*alert* finale, che specifica a quale file appartiene:

```
(function() {
  var a = 0;
  var b = 0;
  var c = 0;
  for(a=0; a<1000; a++)
    for(b=a; b<200000; b++)
      c = a*b-a+b;
  alert('finito script esterno file1.js'); // istruzione che varia
}());
```

Si noti come viene costruita una funzione (anonima), affinché tutte le variabili usate siano locali a essa e non ci siano conflitti, e subito dopo la sua dichiarazione c'è la sua invocazione.
Se si scrive la seguente pagina HTML:

```
<!DOCTYPE html>
<html>
<head>
<script type="text/javascript" src="file.js"></script>
<script type="text/javascript" src="file2.js"></script>
<script type="text/javascript" src="file3.js"></script>
<script type="text/javascript">
  alert('Script dentro la pagina');
</script>
</head>
<body>
</body>
</html>
```

si ha, come risultato, che il caricamento della pagina si ferma per alcuni secondi, poi viene eseguita l'*alert* che si trova in fondo a *file.js*, poi quella dentro *file2.js*, quella su *file3.js* e, infine, riprende l'esecuzione della pagina che esegue l'*alert* che scrive `'Script dentro la pagina'`.

Per quanto concerne la possibilità di caricare porzioni di codice JavaScript in maniera asincrona, si tratta di creare un nuovo elemento di tipo script, con opportuni valori di attributi (in particolare `src` per l'URL del codice JavaScript da caricare) e, novità, impostare l'attributo `async` a **`true`**. In questo modo il file verrà caricato, ma in modalità asincrona, senza bloccare l'utente in attesa del termine della sua esecuzione.

ESEMPIO

Si riscrive la pagina precedente in modo che i file esterni vengano caricati in maniera asincrona:

```
<!DOCTYPE html>
<html>
<head>
<script type="text/javascript">

function getScriptAsync(nome){
  var scriptElem = document.createElement('script');
  scriptElem.type = 'text/javascript';
  scriptElem.src = nome;
  scriptElem.async = true;
  //scriptElem.deferred=true;
  var s = document.getElementsByTagName('script')[0];
  s.parentNode.insertBefore(scriptElem, s);
}
getScriptAsync('file.js');
getScriptAsync('file2.js');
getScriptAsync('file3.js');

</script>

<script type="text/javascript">
  alert('Script dentro la pagina');
</script>
</head>
<body>
  Test asincrono con tre file esterni!
</body>
</html>
```

Eseguendo questa pagina si può notare che non è garantito l'ordine con cui compaiono le *alert* perché l'esecuzione è contemporanea (o comunque secondo un ordine definito internamente dal browser), ma soprattutto l'esecuzione complessiva è molto più veloce, perché non si attende la fine di uno script per passare all'altro.

4.3 Le Google *Maps*

Google ha reso disponibile un servizio, chiamato **Google *Maps***, che permette di visualizzare mappe geografiche e di mostrarle attraverso viste diversificate.
Per gli sviluppatori tali mappe sono rese disponibili per essere incluse nei propri siti web; esse sono personalizzabili in tanti modi, ciascuno dei quali è reso fruibile attraverso opportune API. In questo capitolo si descriverà come utilizzare la versione 3 di tali API (http://code.google.com/apis/maps/index.html).

La prima mappa

Le mappe sono generate dinamicamente dalle API, come pure dinamico è l'aggiornamento del contenuto in base alle azioni dell'utente. La pagina

Quale documentazione

Google mette a disposizione molta documentazione sulle proprie API. Tra i documenti resi disponibili ci sono il *reference*, un *tutorial* e una *developer's guide*.

Il *tutorial* è fatto in modo che, eseguendo passo a passo quanto scritto, si possa arrivare a scrivere semplici esempi funzionanti.

La *developer's guide* affronta un po' più nel dettaglio i contenuti perché è orientata a programmatori.

Infine il *reference* racchiude i dettagli di tutti i metodi e tutti gli attributi offerti dai vari oggetti. Quest'ultimo documento è utile per far fronte a domande specifiche e mirate, per ovviare al fatto che la documentazione completa possa risultare troppo dispersiva.

HTML di partenza deve prevedere pochi e semplici elementi; in particolare deve essere inclusa la libreria delle API. Per farlo si può utilizzare questa dichiarazione nello header del documento.

ESEMPIO

```
<script type="text/javascript"
  src="http://maps.googleapis.com/maps/api/js?sensor=false">
</script>
```

OSSERVAZIONE Si noti il parametro `sensor` in fondo all'URL che permette di caricare la libreria. Esso indica se è presente un sensore per la posizione dell'utente che fa uso della mappa. Per browser che sono eseguiti su PC normali il valore è **`false`**; se invece sono eseguiti su dispositivi che hanno un sensore di posizione (come può essere un dispositivo dotato di GPS) allora va indicato il valore **`true`**. A ogni modo è obbligatorio specificare il parametro e si evidenzia come queste API siano fortemente orientate ai dispositivi mobili.

Ovviamente deve essere prevista un'area dove la mappa verrà visualizzata. Siccome la visualizzazione dell'immagine della mappa sarà successiva al caricamento del documento di partenza, è importante specificare la dimensione dell'area. La mappa non farà altro che riempire tutta quest'area assegnata.

ESEMPIO

```
<body>

  <h1>La prima mappa</h1>
  <div id="mappa" style="width:640px; height:350px"></div>

</body>
```

OSSERVAZIONE Se anziché usare un *div* a dimensione fissa se ne usa uno con grandezza che è una percentuale rispetto al contesto (per esempio 100%), quando si ridimensiona la finestra anche la mappa viene aggiornata ricaricandola dal server e andando a occupare la nuova dimensione come indicato dal valore percentuale. L'utilizzo di `width` e `height` al 100% è raccomandato per dispositivi mobili, dove l'area di visualizzazione della mappa è di per sé ridotta, tanto che è preferibile non ridurla ulteriormente, in modo da poter mostrare una mappa il più possibile leggibile.

La mappa va creata specificando prima le caratteristiche che deve avere, indicate come proprietà di un oggetto passato al costruttore. Il nome e il significato delle principali proprietà sono mostrati nella TABELLA **10**.

TABELLA **10**

Proprietà	Significato
`mapTypeId`	Indica il tipo di visualizzazione; tra i valori ammessi quelli di uso più frequente sono: • google.maps.*Map*TypeId.ROADMAP per le mappe con le strade e altre informazioni cartografiche; • google.maps.*Map*TypeId.SATELLITE per le foto satellitari; • google.maps.*Map*TypeId.HYBRID per le foto satellitari con in più le principali informazioni della modalità ROADMAP; • google.maps.*Map*TypeId.TERRAIN con le informazioni altimetriche per le montagne e la profondità dei mari. È una proprietà che è obbligatorio specificare.
`center`	Le coordinate del centro della mappa. Anche per questa proprietà è obbligatorio specificare un valore. Tale valore è un oggetto di tipo google.maps.LatLng. Per crearlo si usa un costruttore con due parametri che sono, rispettivamente, la latitudine e la longitudine del punto.
`zoom`	Livello di zoom (dove 0 è «nessun zoom» e corrisponde all'immagine dell'intero globo, numeri maggiori corrispondono a zoom maggiori, visualizzando porzioni di mappa più dettagliati).

Ecco una funzione che crea una mappa centrata sul Colosseo a Roma con un valore di zoom pari a 8.

ESEMPIO

```
<script type="text/javascript">
    var map;
    function inizializzaMappa() {
      var myOptions = {
        zoom: 18,
        center: new google.maps.LatLng(41.8905, 12.4921),
        mapTypeId: google.maps.MapTypeId.SATELLITE
      };
      map = new
        google.maps.Map(
          document.getElementById('mappa'),
          myOptions
        );
    }
  </script>
```

Questa funzione va richiamata affinché crei effettivamente la mappa e la disegni a video. Il modo migliore per richiamarla è quello di aspettare il caricamento del documento, ovvero associarlo all'evento `load`. Ecco pertanto la pagina completa risultante e, in FIGURA **12**, la mappa generata.

FIGURA **12**

ESEMPIO

```
<!DOCTYPE html>
<html>
     <head>
   <script type="text/javascript" src="http://maps.googleapis.com/maps/api/js?sensor=false">
   </script>
<script type="text/javascript">
          var map;
          function inizializzaMappa() {
            var myOptions = {
               zoom: 18,
               center: new google.maps.LatLng(41.8905, 12.4921),
               mapTypeId: google.maps.MapTypeId.SATELLITE
            };
            map = new
               google.maps.Map(
                  document.getElementById('mappa'),
                  myOptions
               );
          }
       </script>
     </head>
   <body onload="inizializzaMappa()">
     <h1>La prima mappa</h1>
     <div id="mappa" style="width:640px; height:350px"></div>
</body>
</html>
```

Georeferenziare un indirizzo

Il primo esempio mette subito in evidenza che per poter mostrare una mappa dettagliata di un luogo è necessario conoscerne le coordinate. Questo aspetto non è affatto scontato e ci si potrebbe chiedere come risolverlo. Google mette a disposizione ulteriori API che permettono, dato un indirizzo, di conoscerne le coordinate: queste sono chiamate ***Google Geocoding API*** (http://code.google.com/apis/maps/documentation/geocoding/).

A chi utilizza le Google *Maps* non serve ricorrere a questo servizio, ma si può far uso di una classe che, dietro le quinte, lo utilizza. La classe è `geocoder`. Tale classe vuole, in ingresso, un oggetto con impostata la proprietà `address` che contiene l'indirizzo da georeferenziare; inoltre il risultato è un array di oggetti fatti in questo modo (è un array perché, cercata una stringa di risultati, può essercene più di uno).

ESEMPIO

```
{
     types[]: string,
     formatted_address: string,
     address_components[]: {
       short_name: string,
       long_name: string,
       types[]: string
     },
```

```
    geometry: {
      location: LatLng,
      location_type: GeocoderLocationType
      viewport: LatLngBounds,
      bounds: LatLngBounds
    }
  }
```

Le prime proprietà sono quasi sempre di interesse per una persona che «legge» i risultati e li vuole interpretare correttamente; infatti *types* è un array di stringhe ciascuna delle quali identifica se il risultato è una città, un'entità politica e così via. La proprietà `formatted_address` è l'indirizzo cercato ma normalizzato; tale normalizzazione viene «espansa» nei componenti che lo costituiscono sull'array `address_components`. La successiva proprietà, `geometry`, permette di conoscerne la localizzazione vera e propria. Essa è a sua volta un oggetto di cui interessa, in questo contesto, la proprietà `location` che contiene le coordinate di latitudine e longitudine.

ESEMPIO

Ecco l'esempio precedente affinché l'utente possa specificare un indirizzo e georeferenziarlo. Il risultato dello georeferenziazione viene riportato in vari campi di una form.

```
<!DOCTYPE html>
<html>
     <head>
  <script type="text/javascript" src="http://maps.googleapis.com/maps/api/js?sensor=false">
  </script>
<script type="text/javascript">
          var map;
          function inizializzaMappa() {
            var myOptions = {
              zoom: 18,
              center: new google.maps.LatLng(41.8905, 12.4921),
              mapTypeId: google.maps.MapTypeId.SATELLITE
            };
            map = new
              google.maps.Map(document.getElementById('mappa'),myOptions);
          }
            function georeferenzia(){
              var indirizzo = document.getElementById('indirizzo').value;
              var geocoder = new google.maps.Geocoder();
            geocoder.geocode({'address': indirizzo}, funzioneCallback);
            }

          function funzioneCallback(risultati, status) {
            if (status == google.maps.GeocoderStatus.OK) {
              var coord = risultati[0].geometry.location;
```

```
            map.setCenter(coord);
            document.getElementById('latitudine').value = coord.lat();
            document.getElementById('longitudine').value = coord.lng();
          }else alert("Errore:"+ status);
        }
      </script>
    </head>
  <body onload="inizializzaMappa()">
  <h1>La prima mappa</h1>
  Indirizzo
  <input type="text" id="indirizzo" name="indirizzo" />
  <input type="button"value="GeoReferenzia" onclick="georeferenzia()" />
  <div id="mappa" style="width:640px; height:350px"></div>
  Lat:<input type="text" id="latitudine" name="latitudine" />
  Lng:<input type="text" id="longitudine" name="longitudine" />
</body>
</html>
      </script>
    </head>
  <body onload="inizializzaMappa()">
    <h1>La prima mappa</h1>
    <div id="mappa" style="width:640px; height:350px"></div>
</body>
</html>
```

Nella FIGURA **13** è mostrata l'esecuzione della georeferenziazione dell'indirizzo «piazza dei miracoli, pisa».

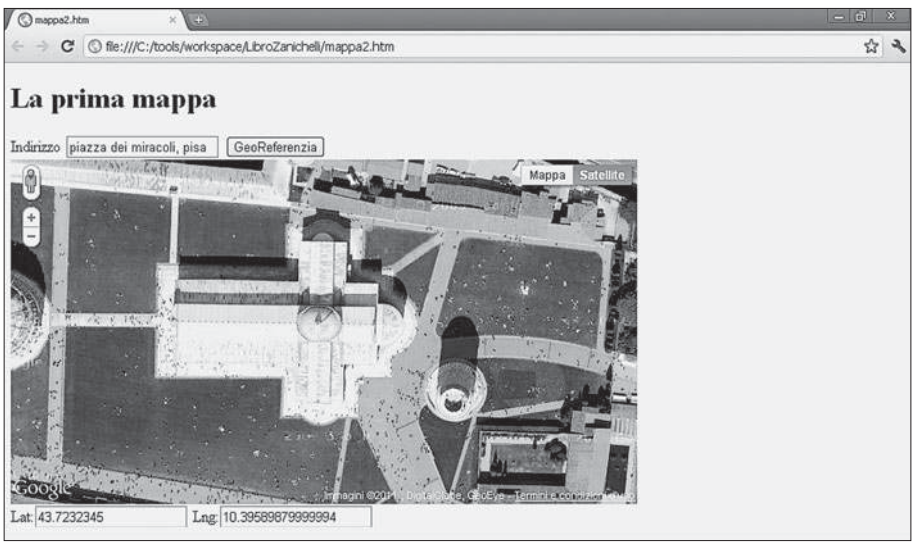


FIGURA **13**

Il servizio di georeferenziazione di Google funziona non solo per gli indirizzi, ma anche per posti noti, come può essere «vesuvio, napoli». In FIGURA **14** è visibile il risultato di una simile ricerca.

FIGURA **14**

Creare un marcatore

Una delle caratteristiche più flessibili e interessanti delle Google *Maps* è quella di poter creare oggetti, figure geometriche e altri elementi direttamente sopra le mappe. Tra gli elementi più utilizzati rientrano i cosiddetti **marcatori**: sono icone posizionate su un punto preciso e che aiutano a indicare, visivamente, una certa posizione. Essi poi possono rispondere agli eventi, come al click su di essi, mostrando fumetti o altri effetti grafici. Un marcatore può essere creato usando il costruttore *Marker*, a cui viene passato un oggetto le cui proprietà possono essere tra quelle mostrate nella TABELLA **11**.

TABELLA **11**

Proprietà	Significato
`map`	La mappa su cui mostrare il marcatore
`position`	Le coordinate dove deve essere mostrato
`title`	Descrizione che compare quando il cursore del mouse è sopra il marcatore
`zIndex`	Quando due o più marcatori si sovrappongono viene mostrato davanti quello con valore `zIndex` più alto. In mancanza di questa proprietà vengono mostrati secondo l'ordine di posizionamento in verticale: quelli più in basso sono mostrati davanti a quelli più in alto
`clickable`	Se `true`, ed è il valore di default, l'oggetto risponde ai click del mouse
`draggable`	Se `true`, può essere spostato. Il default è `false`
`flat`	Se `true`, non viene mostrata l'ombra del marcatore
`cursor`	Tipo di cursore da mostrare quando il mouse è sopra il marcatore
`icon`	Icona da usare come primo piano
`shadow`	Icona da usare come ombra
`animation`	Eventuale animazione da associare all'aggiunta del marcatore a una mappa

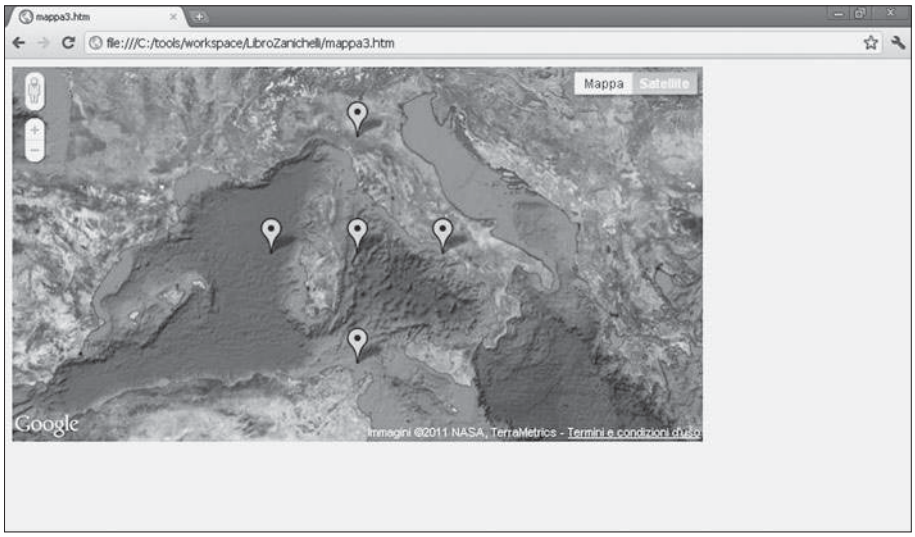


FIGURA **15**

Nella FIGURA **15** è riportata una mappa, con un livello di zoom appropriato, che mostra cinque marcatori, disposti a forma di croce.

ESEMPIO

Il codice che crea i cinque marcatori, a forma di croce e posizionati all'interno del Colosseo. La distanza è specificata nella variabile `d`:

```
<!DOCTYPE html>
<html>
     <head>
  <script type="text/javascript" src="http://maps.googleapis.com/maps/api/js?sensor=false">
  </script>
<script type="text/javascript">
          var map;
          function inizializzaMappa() {
               var lat = 40.8905;
                 var lng = 10.4921;
                 var d = 3.5;
            var myOptions = {
              zoom: 5,
              center: new google.maps.LatLng(lat, lng),
              mapTypeId: google.maps.MapTypeId.SATELLITE
            };
            map = new google.maps.Map(
              document.getElementById('mappa'),myOptions);
                new google.maps.Marker(
                  {position:new google.maps.LatLng(lat, lng), map: map}
                );
                new google.maps.Marker(
                  {position:new google.maps.LatLng(lat+d, lng), map: map}
                );
                new google.maps.Marker(
                  {position:new google.maps.LatLng(lat-d, lng), map: map}
                );
```

▶

```
                new google.maps.Marker(
                  {position:new google.maps.LatLng(lat, lng+d), map: map}
                );
                new google.maps.Marker(
                  {position:new google.maps.LatLng(lat, lng-d), map: map}
                );
            }

      </script>
    </head>
  <body onload="inizializzaMappa()">
    <div id="mappa" style="width:640px; height:350px"></div>
  </body>
</html>
```

Gestire gli eventi

Per le mappe è possibile gestire due tipi di eventi: i primi, simili a quelli del DOM, si riferiscono a determinate azioni sulle mappe, come il click o il movimento del cursore del mouse; altri si riferiscono a proprietà che sono variate in seguito ad azioni dell'utente sull'interfaccia come, per esempio, il cambio di zoom o il tipo di mappa visualizzata. Per poterli gestire si deve associare delle funzioni a determinati eventi; per farlo si usa il metodo `addListener`.

OSSERVAZIONE Benché certi eventi siano simili a quelli del DOM, in realtà sono eventi gestiti dalle librerie di Google *Maps* con caratteristiche proprie. Per esempio è possibile sapere le coordinate del punto dove l'utente ha fatto click.

ESEMPIO

Il metodo `addListener` assume questa forma:

```
google.maps.event.addListener(
     mappa, nome_evento, function(event) {
       // . . .
  });
```

Il primo argomento è la mappa su cui registrare la funzione, il `nome_evento` l'evento a cui rispondere e, infine, la funzione da invocare al verificarsi dell'evento. L'evento è un oggetto passato come argomento alla funzione; tra le sue proprietà anche le coordinate del punto dove si è verificato l'evento: `event.latLng`.

ESEMPIO

```
<!DOCTYPE html>
<html>
     <head>
  <script type="text/javascript" src="http://maps.googleapis.com/maps/api/js?sensor=false">
  </script>
<script type="text/javascript">
```

```
        var map;
        function inizializzaMappa() {
            var myOptions = {
            zoom: 5,
            center: new google.maps.LatLng(40.9, 10.5),
            mapTypeId: google.maps.MapTypeId.SATELLITE
          };
          map = new google.maps.Map(
            document.getElementById('mappa'),myOptions);
        google.maps.event.addListener(
            map, 'click',
              function(event) {
                alert('lat:'+ event.latLng.lat() +
                  ', lng:'+ event.latLng.lng());
            });
        }
      </script>
    </head>
  <body onload="inizializzaMappa()">
    <div id="mappa" style="width:640px; height:350px"></div>
  </body>
</html>
```

Creare una finestra di informazioni

Un oggetto di tipo `google.maps.InfoWindow` è una finestra dalla forma di fumetto che si può aprire all'interno di una mappa. Normalmente la si associa a qualche evento su marcatori, ma è possibile ancorarla su una posizione qualsiasi di una mappa. Il costruttore riceve in ingresso un oggetto su cui è possibile definire le diverse proprietà che deve avere. Nella TABELLA **12** sono illustrate le principali proprietà.

TABELLA **12**

Proprietà	Significato
`content`	Il contenuto da mostrare quando la finestra è aperta. Può essere una porzione di HTML qualsiasi
`position`	Contiene la posizione dove ancorare la finestra. Se essa viene associata a un marcatore, eredita da esso questo valore
`maxWidth`	Massima dimensione della finestra. Se non viene specificato, essa si ridimensiona in base a quanto scritto in `content`

ESEMPIO

Ecco come usare una mappa in cui esiste un marcatore e, al click su di esso, si apre una finestra di informazioni (FIGURA **16**).

```
<!DOCTYPE html>
<html>
    <head>
  <script type="text/javascript"
    src="http://maps.googleapis.com/maps/api/js?sensor=false">
```

```
  </script>
<script type="text/javascript">
        var map;
        function inizializzaMappa() {
            var coord=new google.maps.LatLng(41.8905, 12.4921);
          var myOptions = {
            zoom: 18,
            center: coord,
            mapTypeId: google.maps.MapTypeId.SATELLITE
          };
          map = new google.maps.Map(
            document.getElementById('mappa'),myOptions);
          var infowindow = new google.maps.InfoWindow({
            content: "Un esempio di InfoWindow"
          });

          var marker = new google.maps.Marker({
            position: coord,
            map: map,
            title: 'Titolo'
          });
      google.maps.event.addListener(marker, 'click', function() {
        infowindow.open(map,marker);
      });   }
      </script>
    </head>
  <body onload="inizializzaMappa()">
    <div id="mappa" style="width:640px; height:350px"></div>
  </body>
</html>
```

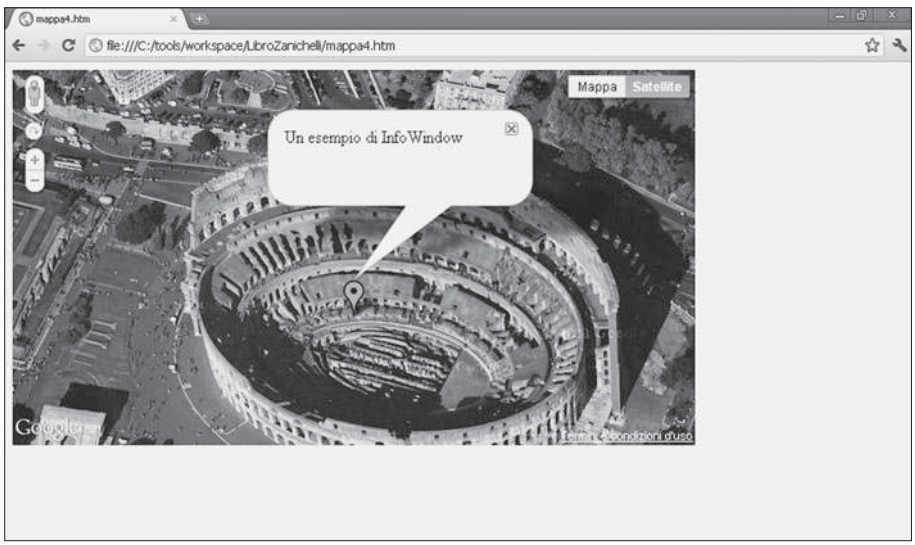


FIGURA **16**

Sintesi

Ogni documento HTML può essere rappresentato con una struttura ad albero. Esistono opportuni metodi per attraversare e modificare l'albero di un documento HTML.

Vari componenti di una pagina possono prevedere **gestori di eventi**. Tali eventi sono legati o al documento stesso (come il caricamento completo di una pagina o di un'immagine) e alle iterazioni con l'utente (cambio focus, cambio contenuto, click con il mouse e così via).

Il **DOM** è una collezione di oggetti che modellano vari aspetti di una pagina. È possibile accedere al DOM grazie a opportune API.

L'oggetto **window** modella la finestra corrente, **navigator** il browser, **location** l'indirizzo della pagina, **history** la cronologia e **screen** le informazioni sul display dell'utente.

È possibile accedere e modificare gli attributi CSS direttamente da JavaScript.

jQuery è una libreria che semplifica lo sviluppo di applicazioni web e fornisce delle primitive interoperabili per accedere ai diversi oggetti del DOM.

Il modello di esecuzione del browser è **single thread**, a meno che non ci siano invocazioni asincrone.

Le **Google *Maps*** permettono di includere mappe navigabili nelle proprie pagine.

È possibile **georeferenziare** un qualsiasi indirizzo grazie ai servizi di georeferenziazione.

Una mappa può essere personalizzata con **marcatori** e **finestre di informazione**.

QUESITI

1 **Per DOM si intende …**

A … un insieme di oggetti esposti da un browser per interagire con il browser stesso e le pagine visualizzate.

B … un linguaggio di programmazione con cui si possono scrivere programmi.

C … una libreria fortemente ottimizzata e che va inclusa nella propria pagina.

2 **Una pagina HTML è rappresentabile come …**

A … un albero.

B … una coda.

C … un grafo completamente connesso.

D Nessuno dei precedenti.

3 **Il metodo `document.getElementById` serve a …**

A … recuperare tutti gli elementi che sono identificati da un certo tipo di tag.

B … recuperare un elemento specificando il suo ID.

C … modificare l'ID di un elemento.

D … modificare il tipo di tag.

4 **Il gestore `onblur` …**

A … viene attivato quando l'elemento perde il focus.

B … viene attivato quando l'elemento viene rimosso.

C … viene attivato quando l'elemento riceve un click.

D Non esiste un simile gestore di evento.

5 **Il gestore `onremoved` …**

A … viene attivato quando l'elemento perde il focus.

B … viene attivato quando l'elemento viene rimosso.

C … viene attivato quando l'elemento riceve un click.

D Non esiste un simile gestore di evento.

6 **Indicare quale tra questi oggetti permette, grazie alle sue proprietà, di conoscere il tipo di browser che sta visualizzando la pagina.**

A `window`

B `history`

C `navigator`

D `location`

E Nessuna delle risposte precedenti.

7 **Utilizzando l'oggetto `location`, che differenza c'è tra usare il metodo `replace` o assegnare alla proprietà `href` un nuovo valore, per cambiare pagina?**

A Non c'è alcuna differenza.

B Si deve usare `replace` perché `href` è una proprietà in sola lettura.

C Usando `replace` non resta traccia della pagina precedente nella cronologia.

D `replace` permette di ricaricare solo la pagina corrente.

E Nessuna delle risposte precedenti.

8 **Supponendo che esista la form con nome `'test'`, qual è il modo corretto per riferirsi a essa?**

A `document.getElementById('test')`

B `document.forms['test']`

C `window.location['test']`

D Nessuna delle risposte precedenti.

9 **È possibile cambiare il valore delle proprietà CSS direttamente da JavaScript?**

A No, sono due mondi indipendenti.

B Sì, solo se la proprietà non è stata precedentemente impostata da un foglio di stile.

C Sì, usando `oggetto.style.nomeProprietà`

D Nessuna delle risposte precedenti.

10 **Indicare quale delle seguenti non è una libreria JavaScript.**

A jQuery.

B YUI Library.

C Prototype.

D W3C.

11 **In jQuery, come si fa a selezionare un elemento con ID `'nomeId'`?**

A `$("!nomeId")`

B `$("#nomeId")`

C `$("@nomeId")`

D Nessuna delle risposte precedenti.

12 **A cosa serve il metodo `val()` di jQuery?**

A A sapere il valore del colore dell'elemento a cui si applica.

B A sapere il valore del campo della form a cui si applica.

C A sapere il valore della posizione all'interno della pagina dell'emento a cui si applica.

D Nessuna delle risposte precedenti.

13 **Se in una porzione di codice JavaScript c'è un loop infinito …**

A … è una situazione non possibile: qualunque loop termina sicuramente, per definizione di loop.

B … il browser si blocca, a meno che non abbia controlli sul tempo di esecuzione del codice JavaScript, e propone all'utente di fermare lo script se la sua esecuzione è troppo lunga.

C … il browser continua il rendering del resto della pagina e l'esecuzione di eventuali altri script; in pratica non ci si accorge che una parte del codice resta in esecuzione senza terminare.

D Nessuna delle situazioni precedenti.

14 **Indicare qual è il costruttore per creare una mappa in Google *Maps*.**

A `new google.mapa.Map(proprietà)`

B `new google.mapa.Map(proprietà1, proprietà2, …)`

C `new google.mapa.Map()`

D Nessuna delle situazioni precedenti.

15 **Cosa permette di rappresentare un oggetto del tipo `google.maps.Marker`?**

A Un supermercato.

B Uno strumento per lo zoom attraverso una scala graduata.

C Un marcatore sulla mappa.

D Nessuna delle situazioni precedenti.

16 **Gli eventi gestite nelle Google *Maps* sono …**

A … esattamente gli stessi del DOM.

B … alcuni sono simili a quelli del DOM «arricchiti» con informazioni specifiche e altri eventi legati ai controlli sulle mappe.

C … le Google *Maps* non gestiscono gli eventi.

D Nessuna delle situazioni precedenti.

ESERCIZI

1 Spiegare la differenza dei gestori `onkeypressed` e `onkeydown`, anche con un esempio pratico.

2 Scrivere una pagina HTML con un campo di testo e un pulsante «controlla». Quando l'utente preme il pulsante mostra una *alert* con `"Campo vuoto"` se il contenuto del campo di testo non contiene alcun valore, altrimenti mostra il valore contenuto.

3 Scrivere una funzione che legge tutti i tag `<p>` e inserisce, prima del contenuto, un numero progressivo tra parentesi quadre. Così il contenuto del primo `<p>` inizierà con «[1]», il secondo con «[2]» e così via.

4 Creare uno script che colleziona tutti i link di un documento e crea una «sitografia» automatica in coda alla pagina.

5 Quali possono essere i vantaggi di utilizzare una libreria come jQuery? Elencare almeno tre elementi a favore, motivando le proprie scelte.

6 Con jQuery, trovare tutte le tabelle di una pagina e aggiungervi la classe CSS «tabella».

7 Realizzare una Google *Map* che intercetta il click dell'utente e, in quel punto, crea un nuovo marcatore.

8 Modificare la soluzione dell'esercizio precedente affinché, oltre al marcatore, compaia anche un fumetto contenente le informazioni sulle coordinate (latitudine e longitudine) del punto in cui l'utente ha fatto click.

LABORATORIO

1 Si crei una pagina HTML che, grazie al codice JavaScript, permetta di giocare al gioco dell'impiccato. In particolare, si faccia in modo di modellare correttamente l'intero gioco e l'interfaccia grafica, prestando attenzione ai seguenti aspetti:

a) permettere la scelta pseudocasuale della parola da indovinare partendo da un dizionario di un certo numero di parole;

b) gestire la visualizzazione della parola, in modo che inizialmente compaiano solo la prima e l'ultima lettera, e un trattino o un carattere di «underscore» per le lettere intermedie;

c) semplificare l'interfaccia affinché l'utente possa scegliere una delle lettere dell'alfabeto da inserire nella parola da indovinare;

d) mostrare l'immagine dell'impiccato, che viene aggiornata dopo ogni errore del giocatore.

2 Si crei una pagina HTML con una Google *Map* e una textarea. Fare in modo che ogni volta che l'utente fa click su un punto della mappa, su tale punto venga visualizzato un nuovo marcatore e, contestualmente, nella textarea venga aggiunto il codice JavaScript per la creazione del marcatore stesso. In questo modo, al termine dei click, l'utente potrebbe fare copia e incolla del contenuto della textarea e inserirlo nel file HTML della mappa, al fine di rendere persistenti i marcatori.

What is the Document Object Model?

Editors
Jonathan Robie, Texcel Research

Introduction

The Document Object Model (DOM) is an application programming interface (API) for valid HTML and well-formed XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. In the DOM specification, the term "document" is used in the broad sense - increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data.

With the Document Object Model, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the Document Object Model, with a few exceptions - in particular, the DOM interfaces for the XML internal and external subsets have not yet been specified.

As a W3C specification, one important objective for the Document Object Model is to provide a standard programming interface that can be used in a wide variety of environments and applications. The DOM is designed to be used with any programming language. In order to provide a precise, language-independent specification of the DOM interfaces, we have chosen to define the specifications in Object Management Group (OMG) IDL [OMGIDL], as defined in the CORBA 2.2 specification [CORBA]. In addition to the OMG IDL specification, we provide language bindings for Java [Java] and ECMAScript [ECMAScript] (an industry-standard scripting language based on JavaScript and JScript).

[…]

What the Document Object Model is

The DOM is a programming API for documents. It is based on an object structure that closely resembles the structure of the documents it models. For instance, consider this table, taken from an HTML document:

```
<TABLE>
  <TBODY>
    <TR>
      <TD>Shady Grove</TD>
      <TD>Aeolian</TD>
    </TR>
    <TR>
      <TD>Over the River, Charlie</TD>
      <TD>Dorian</TD>
    </TR>
  </TBODY>
</TABLE>
```

The DOM represents this table as below:

API
Application Programming Interface: insieme di procedure disponibili al programmatore, riutilizzabili per lo sviluppo di applicazioni

well-formed
ben formato, «sintatticamente corretto»

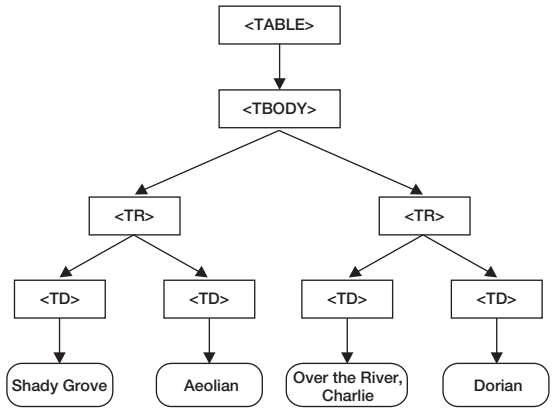
XML
eXtensible Markup Language: metalinguaggio di markup

nevertheless
ciò non di meno

CORBA
Common Object Request Broker Architecture: standard sviluppato da OMG per permettere la comunicazione fra componenti eterogenei sulla rete

grove
Piccolo bosco senza sottobosco

ORIGINAL DOCUMENT



In the DOM, documents have a logical structure which is very much like a tree; to be more precise, it is like a "forest" or "grove", which can contain more than one tree. Each document contains zero or one doctype nodes, one root element node, and zero or more comments or processing instructions; the root element serves as the root of the element tree for the document. However, the DOM does not specify that documents must be implemented as a tree or a grove, nor does it specify how the relationships among objects be implemented. The DOM is a logical model that may be implemented in any convenient manner. In this specification, we use the term structure model to describe the tree-like representation of a document. We also use the term "tree" when referring to the arrangement of those information items which can be reached by using "tree-walking" methods; (this does not include attributes). One important property of DOM structure models is structural isomorphism: if any two Document Object Model implementations are used to create a representation of the same document, they will create the same structure model, in accordance with the XML Information Set.

QUESTIONS

a What is the Document Object Model (DOM)?

b What can be accessed, changed, deleted, or added using the DOM?

c Which markup languages is the DOM applicable to?

d What language bindings does W3C provide for the DOM?

Indice analitico

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Fiorenzo Formichi Giorgio Meini Ivan Venuti

Corso di informatica

per Informatica

Programmazione orientata agli oggetti e linguaggio Java
Pagine web con JavaScript

Un corso di informatica focalizzato sulla pratica di laboratorio, mediante esempi graduali e attinenti ad aspetti professionalmente rilevanti. I contenuti proposti e gli esempi applicativi sono illustrati attraverso i linguaggi di programmazione più diffusi nella pratica professionale.

Nel libro

- Il testo è diviso in **due sezioni**: la prima dedicata a un tema fondamentale (*Programmazione orientata agli oggetti e linguaggio Java*), la seconda a una tematica web (*Pagine web con JavaScript*).
- Una **sintesi di fine capitolo** introduce i quesiti e gli esercizi.
- **Brani in inglese**, tratti da testi di riferimento della disciplina, sono accompagnati da un glossario.

NOVITÀ 2012