

**Thinking in Java**

---

Concorrenza  
e interfacce grafiche



Bruce Eckel

# **Thinking in Java**

## **Concorrenza e interfacce grafiche**

Quarta edizione



Copyright © 2006 Pearson Education Italia S.r.l.  
Via Fara, 28 - 20124 Milano  
Tel. 02/6739761 - Fax 02/673976503  
E-mail: hpeitalia@pearson.com  
Web: <http://hpe.pearsoned.it>

*Authorized translation from the English language edition, entitled: THINKING IN JAVA, 4<sup>th</sup> Edition, 0131872486 by Eckel, Bruce, published by Pearson Education, Inc, publishing as Prentice Hall PTR Copyright © 2006.*

*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 Education Italia Srl, Copyright © 2006.*

Le informazioni contenute in questo libro sono state verificate e documentate con la massima cura possibile. Nessuna responsabilità derivante dal loro utilizzo potrà venire imputata agli Autori, a Pearson Education Italia o a ogni persona e società coinvolta nella creazione, produzione e distribuzione di questo libro.

I diritti di riproduzione e di memorizzazione elettronica totale e parziale con qualsiasi mezzo, compresi i microfilm e le copie fotostatiche, sono riservati per tutti i paesi.

**LA FOTOCOPIATURA DEI LIBRI È UN REATO.**

L'editore potrà concedere a pagamento l'autorizzazione a riprodurre una porzione non superiore a un decimo del presente volume. Le richieste di riproduzione vanno inoltrate ad AIDRO (Associazione Italiana per i Diritti di Riproduzione delle Opere dell'Ingegno), Via delle Erbe, 2 - 20121 Milano - Tel. e Fax 02/80.95.06.

Traduzione: Georges Piriou, Marco Tripolini  
Revisione tecnica: Georges Piriou, Marco Tripolini  
Realizzazione editoriale: Art Servizi Editoriali - Bologna  
Grafica di copertina: Sabrina Miraglia

Stampa: Presscolor - Milano

Tutti i marchi citati nel testo sono di proprietà dei loro detentori.

ISBN 13: 978-8-8719-2305-5  
ISBN 10: 88-7192-305-7

Printed in Italy

1<sup>a</sup> edizione: settembre 2006

## Struttura dell'opera

	Vol. 1	Vol. 2	Vol. 3
Prefazione	•	•	•
Introduzione	•	•	•
Introduzione agli oggetti	•		
Tutto è un oggetto	•		
Gli operatori	•		
Il controllo dell'esecuzione	•		
Inizializzazione e cleanup	•		
Controllo di accesso	•		
Riutilizzo delle classi	•		
Il polimorfismo	•		
Le interfacce	•		
Le classi interne	•		
Come contenere gli oggetti	•		
Gestione degli errori con le eccezioni		•	
Le stringhe	•		
Informazioni sui tipi		•	
I generici		•	
Gli array		•	
Ancora sui contenitori		•	
Input/Output		•	
I tipi enumerativi		•	
Le annotazioni		•	
La concorrenza			•
Interfacce grafiche (GUI)			•
Supplementi			•
Risorse	•	•	•

L'edizione italiana presenta, rispetto a quella anglosassone, alcune importanti modifiche, che hanno portato alla suddivisione dell'opera originale in tre volumi e ad una parziale riorganizzazione dei contenuti. Lo schema qui sopra riportato illustra sinteticamente la struttura di questa edizione. Sono nati così tre testi autonomi, che speriamo rendano più agevole la consultazione e consentano anche una migliore fruibilità dei contenuti.



# Indice

<b>Prefazione</b>	<b>XIII</b>
Java SE5 e SE6	XV
Java SE6	XV
La quarta edizione	XVI
Modifiche	XVI
Note sulla grafica di copertina	XVIII
Ringraziamenti dell'autore	XIX
<b>Introduzione</b>	<b>XXIII</b>
Prerequisiti	XXIV
Imparare Java	XXIV
Obiettivi	XXV
Imparare da questo libro	XXVI
Documentazione JDK in HTML	XXVII
Esercizi	XXVII
Fondamenti di Java	XXVIII
Codice sorgente	XXVIII
Standard di codifica	XXXI
Errori	XXXI
<b>Capitolo 1 - La concorrenza</b>	<b>1</b>
Molteplici aspetti della programmazione concorrente	3
Velocità di esecuzione	3
Migliorare la progettazione del codice	7
Gestione di base dei thread	8
Definizione dei task	9
Classe Thread	11
Utilizzo di Executor	13
Ottenere valori di ritorno dai task	17
Messa in pausa con sleep()	19
Priorità	21
Yielding	23
Thread demoni	24
Variazioni sul codice	30



Terminologia	37
Collegamento dei thread	38
Creazione di interfacce utente reattive	40
Gruppi di thread	42
Intercettazione delle eccezioni	43
Condivisione delle risorse	46
Accesso improprio alle risorse	47
Risoluzione di conflitti tra risorse condivise	50
Sincronizzazione di EventGenerator	53
Utilizzo di oggetti Lock esplicativi	54
Atomicità e volatilità	57
Classi atomiche	65
Sezioni critiche	67
Sincronizzazione su altri oggetti	74
Memoria locale di thread	76
Chiusura dei task	78
Il giardino ornamentale	78
Terminare un task bloccato	83
Stati dei thread	83
Condizione di blocco	84
Interruzione	84
Blocco causato da mutex	92
Controllo di un interrupt	95
Cooperazione tra task	98
Metodi wait() e notifyAll()	99
Segnali mancanti	105
Confronto tra notify() e notifyAll()	106
Produttori e consumatori	110
Produttori, consumatori e code	118
Utilizzo di BlockingQueue nella produzione di toast!	120
Utilizzo delle pipe per l'I/O tra i task	124
Deadlock	127
Nuovi componenti di libreria	134
CountDownLatch	134
Funzionalità di libreria per la sicurezza dei thread	137
CyclicBarrier	137
DelayQueue	141
PriorityBlockingQueue	145
Un controller di serraglio con ScheduledExecutor	148
Semafori	153
Exchanger	158
Simulazione	161
Simulatore di uno sportello bancario	161
Simulatore di un ristorante	168
Attività distribuite	175
Ottimizzazione delle prestazioni	182



Confronto tra le tecnologie mutex	182
Contenitori non soggetti a lock	193
Problemi di ottimizzazione	194
Confronto tra implementazioni Map	201
Lock ottimistico	203
Lock di tipo ReadWriteLock	206
Oggetti attivi	210
Riepilogo	215
Ulteriori letture	217
<b>Capitolo 2 - Interfacce grafiche (GUI)</b>	<b>219</b>
Gli applet	222
Nozioni di base su Swing	223
Un framework di visualizzazione	226
Costruzione di un pulsante	227
Intercettare un evento	228
Area di testo	232
Controllare la disposizione dei componenti	234
BorderLayout	234
FlowLayout	236
GridLayout	237
GridBagLayout	237
Posizionamento assoluto	238
BoxLayout	238
Qual è l'approccio migliore?	239
Modello a eventi di Swing	239
Tipi di eventi e di listener	240
Semplificazione mediante gli adattatori di listener	246
Monitoraggio di eventi multipli	247
Selezione di componenti Swing	251
Pulsanti	251
Gruppi di pulsanti	253
Icone	255
Tooltip	257
Campi di testo	257
Bordi	260
Un mini-editor	261
Caselle di scelta	263
Pulsanti radio	264
Caselle combinate o elenchi a discesa	266
Caselle di riepilogo	268
Pannelli a schede	270
Finestre di messaggio	271
Menu	274
Menu pop-up	282
Disegno	283



Finestre di dialogo	288
Finestre di dialogo per i file	292
HTML nei componenti Swing	295
Cursori scorrevoli e barre di avanzamento	296
Modificare l'aspetto dell'interfaccia	298
Strutture ad albero, tabelle e appunti	301
JNLP e Java web Start	301
Concorrenza e Swing	307
Task di lunga durata	308
Threading visuale	318
Programmazione visuale e JavaBeans	320
Che cos'è JavaBean?	322
Estrazione di BeanInfo con Introspector	324
Un bean più complesso	331
JavaBeans e sincronizzazione	335
Pacchettizzazione dei bean	340
Supporto ai bean avanzato	342
Ulteriori risorse su JavaBeans	343
Alternative a Swing	343
Costruire client web in Flash con Flex	344
Hello, Flex	345
Compilare MXML	346
MXML e ActionScript	348
Contenitori e controlli	349
Effetti e stili	351
Eventi	352
Connettersi a Java	353
Modelli dei dati e collegamento ai dati	356
Compilazione e distribuzione	357
Creare applicazioni SWT	360
Installazione di SWT	360
Hello, SWT	361
Eliminare il codice ridondante	365
Menu	367
Pannelli a schede, pulsanti ed eventi	369
Grafica	373
Concorrenza in SWT	376
SWT o Swing?	379
Riepilogo	379
Risorse	380
<b>Appendice A - Supplementi</b>	<b>381</b>
Supplementi scaricabili	381
Thinking in C: i fondamenti di Java	382
Seminari Thinking in Java	382
Seminario Hands-On Java su CD	382



Seminario Thinking in Objects	383
Thinking in Enterprise Java	383
Thinking in Patterns (with Java)	384
Seminario Thinking in Patterns	384
Consulenza e revisione di progetti	385

---

<b>Appendice B - Risorse</b>	<b>387</b>
------------------------------	------------

Software	387
Editor e ambienti IDE	388
Libri	388
Analisi e progettazione	389
Python	392
Bibliografia dell'autore	393

---

<b>Indice analitico</b>	<b>395</b>
-------------------------	------------



Thinking in Java

# Prefazione

L'autore di questo manuale si è avvicinato al linguaggio Java ritenendolo "semplicemente un altro linguaggio di programmazione" come in effetti è, sotto molti punti di vista.

Tuttavia, studiando Java in modo più approfondito, ha avuto modo di notare come lo scopo fondamentale di questo linguaggio fosse profondamente diverso da quello degli altri linguaggi conosciuti fino a quel momento.

Programmare significa gestire la complessità: quella del problema da risolvere, cui si aggiunge la complessità del sistema sul quale il problema viene risolto. Per queste ragioni, la maggior parte dei progetti di programmazione non giunge a buon fine. Tra l'altro, di tutti i linguaggi noti all'autore, ben pochi hanno affrontato questo aspetto della programmazione, ritenendo che l'obiettivo principale consistesse nel ridurre drasticamente la complessità dello sviluppo e la manutenzione del software.<sup>1</sup>

Ovviamente molte scelte progettuali nei diversi linguaggi hanno dovuto prendere atto di tale complessità, sebbene, a un certo punto, vi fossero altri elementi essenziali di cui tenere conto: gli stessi elementi che, alla fine, fanno sì che molti programmatore si ritrovino a "sbattere la testa contro il muro". Per esempio, C++ ha dovuto conservare la retrocompatibilità con il C, in modo da garantire una migrazione indolore per i programmatore C e nel contempo mantenere un'efficienza elevata.

Si tratta senza dubbio di considerazioni che hanno contribuito in modo notevole al successo di C++, originando però una maggiore complessità, tale da impedire il completamento di molti progetti. È certamente possibile attribuire la responsabilità ai programmatore o alla direzione aziendale, tuttavia perché non avvantaggiarsi di un particolare linguaggio, se consente di evitare errori di programmazione?

Un altro esempio è fornito da Visual BASIC (VB): legato al BASIC, VB non è stato realmente progettato per essere un linguaggio estensibile, pertanto tutte le estensioni accumulate con il tempo hanno prodotto una sintassi in alcuni casi davvero ingestibile.

---

1. L'autore ritiene, tuttavia, che il linguaggio Python sia quello che più si avvicina a questo obiettivo; si veda [www.python.org](http://www.python.org).



Pur essendo retrocompatibile con *awk*, *sed*, *grep* e altri strumenti Unix per sostituire i quali è stato ideato, Perl viene spesso tacciato di produrre codice cosiddetto "*write-only*", difficilmente leggibile e interpretabile a distanza di tempo.

Di contro, nella progettazione di C++, VB, Perl e altri come SmallTalk, si è considerato almeno in parte il problema della complessità, con il risultato che questi linguaggi si sono dimostrati notevolmente efficaci per risolvere problemi specifici.

Ciò che ha colpito particolarmente l'autore è stato comprendere che, tra gli obiettivi progettuali di Sun, vi sia stato quello di limitare la complessità per il programmatore, quasi a voler affermare: "ci siamo preoccupati di ridurre i tempi e le difficoltà connesse con la produzione di codice affidabile". All'inizio questo ha portato a un codice la cui esecuzione non era particolarmente rapida (problema che si è poi ridimensionato), ma ha anche consentito una considerevole riduzione nei tempi di sviluppo: anche meno della metà del tempo richiesto per creare un analogo programma in C++.

Se questo consente di risparmiare tempo e denaro, Java non si ferma qui. Incorpora numerosi processi completi e importanti (tra cui il multithreading e la programmazione di rete) nel linguaggio e nelle librerie che a volte possono semplificarli. Infine, Java consente di affrontare problemi notevolmente articolati, quali i programmi multipiattaforma, le modifiche dinamiche del codice e la sicurezza, ognuno dei quali occupa, nella "scala di complessità" di ogni programmatore, una posizione variabile da *impedimento a ostacolo insormontabile*. Pertanto, nonostante i problemi di prestazioni, cui si è accennato, le promesse di Java sono considerevoli, poiché può trasformarci in programmatore assai più produttivi.

Comunque sia, Java aumenta la possibilità di comunicazione tra persone: ciò risulta evidente nella realizzazione di programmi, nel lavoro di gruppo, nella produzione di interfacce per comunicare con l'utente, nell'esecuzione di programmi su diversi sistemi e nella scrittura di programmi per comunicare via Internet.

I risultati della rivoluzione nella comunicazione potrebbero talvolta confondersi con gli effetti della movimentazione di grandi quantità di dati. La vera rivoluzione si riconoscerà, invece, perché saremo in grado di comunicare più facilmente: l'uno con l'altro, ma anche in gruppi e globalmente. Alcuni sostengono che tale rivoluzione consista nella formazione di una *mente globale*, risultante da un insieme sufficiente di persone e dotata di sufficienti interconnessioni. Java potrà anche non essere lo strumento che darà il via a una simile rivoluzione, in ogni caso l'esistenza di questa possibilità ha motivato l'autore nell'insegnare questo linguaggio.



## Java SE5 e SE6

Questa edizione del manuale tiene conto dei numerosi miglioramenti apportati al linguaggio Java, in ciò che Sun originariamente ha chiamato JDK 1.5, più tardi diventato JDK5 o J2SE5, e che infine ha perso il “2”, ormai sорpassato, per trasformarsi in Java SE5. Molte modifiche apportate a Java SE5 sono state ideate per migliorare l’attività del programmatore. Come vedrete, i progettisti di Java non sono riusciti a soddisfare appieno questa esigenza, per quanto in generale abbiano compiuto notevoli passi nella direzione giusta.

Uno degli obiettivi principali di questa edizione consiste nel far comprendere tutti i miglioramenti presenti in Java SE5/6, esaminandoli e utilizzandoli nel prosieguo del volume. Ciò implica che questo manuale si assume l’onere, in un certo senso audace, di essere “esclusivo per Java SE5/6”, e che quindi la gran parte del codice non sarà compilabile con le precedenti versioni. Nella fase di build, infatti, Java visualizzerà errori e si interromperà.

Se per qualsiasi motivo siete vincolati a versioni anteriori, tenete presente che anche questa edizione fornisce le basi del linguaggio, e che le versioni precedenti di questo manuale sono comunque disponibili per il download gratuito da [www.mindview.net](http://www.mindview.net). Per varie ragioni, si è deciso di non fornire gratuitamente la versione digitale dell’edizione corrente, ma soltanto quelle precedenti.

## Java SE6

Questo manuale è un progetto imponente che ha richiesto molto tempo. Prima della sua pubblicazione è stata presentata la versione beta di Java SE6 (nome in codice *mustang*). Nonostante la presenza di modifiche secondarie, che hanno consentito di perfezionare alcuni degli esempi presenti nel libro, in genere le nuove caratteristiche di Java SE6 non hanno influito sui contenuti del volume; le novità riguardano soprattutto la velocità di esecuzione e le funzionalità di libreria che non rientrano negli scopi di questo manuale.

Il codice esposto in questo libro è stato testato con successo con una versione *release-candidate* di Java SE6, che non dovrebbe presentare variazioni tali da influire sul contenuto del manuale; qualora la versione definitiva di Java SE6 dovesse essere caratterizzata da modifiche sostanziali, esse saranno riportate nel codice sorgente del manuale, liberamente scaricabile da [www.mindview.net](http://www.mindview.net).

Questo manuale è destinato a Java SE5/6, il che significa che è stato scritto tenendo in considerazione Java SE5 e le modifiche significative introdotte da questa versione, pur essendo ugualmente applicabile anche a Java SE6.

## La quarta edizione

La soddisfazione di realizzare la nuova edizione di un manuale consiste nel poter fare tesoro dell'esperienza acquisita con la precedente, e nell'opportunità di rivedere passaggi difficili o semplicemente noiosi. Come sempre la preparazione di una nuova edizione dà origine a idee affascinanti e inedite, e l'imbarazzo che potrebbe sorgere dalla rilettura del proprio lavoro è attenuato dal piacere di esprimere le proprie idee in una forma sempre migliore.

Ma nella mente dell'autore vi è anche la sfida a produrre un lavoro che perfino i possessori delle edizioni precedenti desiderino acquistare: ciò lo stimola a migliorare, riscrivere e riorganizzare quanto più possibile, per offrire ai lettori una nuova esperienza costruttiva.

### Modifiche

Il CD-ROM che in passato veniva fornito come parte integrante del manuale non è incluso in questa edizione. La parte più importante del CD, il seminario multimediale *Thinking in C*, prodotto per MindView da Chuck Allison, ora è disponibile come presentazione in formato Flash, e scaricabile liberamente in lingua inglese. L'obiettivo di questo seminario è fornire ai lettori che non avessero familiarità con la sintassi del C le competenze indispensabili per comprendere il materiale presentato in questo manuale. Nonostante due capitoli siano dedicati a un'introduzione alla sintassi del linguaggio C, questi potrebbero rivelarsi insufficienti per chi non disponesse della necessaria esperienza: *Thinking in C* è stato progettato espressamente per aiutare questi utenti a conseguire il livello di competenza necessario.

Pur essendo stato interamente riscritto tenendo conto delle principali modifiche intervenute nelle librerie Java SE5 "Concurrency Utilities", il Capitolo 1 del Volume 3, *La concorrenza (Multithreading*, nelle edizioni precedenti in lingua inglese), fornisce ancora i fondamenti teorici della programmazione concorrente. Senza queste basi sarebbe arduo comprendere gli argomenti più complessi relativi al threading. L'autore ha trascorso diversi mesi lavorando su queste caratteristiche di Java, in quel mondo sotterraneo chiamato "programmazione concorrente": ne è risultato un capitolo che, oltre ai fondamenti teorici, si addentra in una trattazione più avanzata.

È presente un nuovo capitolo per ogni nuova caratteristica importante introdotta in Java SE5, mentre altre novità sono state incorporate nel materiale esistente. Grazie all'impegno continuativo dell'autore nello studio dei design pattern, il manuale è stato arricchito di nuovi pattern.



L'autore ha sottoposto il manuale a una profonda riorganizzazione, soprattutto grazie all'esperienza maturata con l'insegnamento e alla consapevolezza che il concetto di "capitolo" meritasse qualche ripensamento. In generale, si è considerato che per meritare un intero capitolo l'argomento dovesse essere sufficientemente corposo. Tuttavia, in particolare nelle lezioni sul design pattern, l'autore ha notato che i partecipanti ai seminari ottengono risultati migliori se si presenta una struttura per volta, immediatamente seguita da un'esercitazione; questo approccio è anche più gratificante per il docente. Per tali motivi in questa versione del manuale i capitoli sono stati suddivisi per argomento, senza tenere conto della loro dimensione. L'autore è convinto che ciò rappresenti un miglioramento.

Grande importanza è stata assegnata anche alla verifica del codice. Senza una struttura di test incorporata che esegue verifiche ognqualvolta si procede alla compilazione di un progetto, non si può in alcun modo valutare l'affidabilità del codice. A questo scopo è stata creata una struttura di test, scritta in linguaggio Python, per visualizzare e convalidare l'output di ogni programma: potete scaricarla dal sito [www.mindview.net](http://www.mindview.net), nel codice relativo a questo manuale. L'argomento dei test in generale è trattato nel supplemento che potrete consultare all'indirizzo <http://mindview.net/Books/BetterJava>.

Sono stati riesaminati anche tutti gli esempi del manuale, con l'obiettivo di rispondere alla domanda: "per quale ragione è stata scelta questa soluzione?". Nella maggior parte dei casi sono state apportate modifiche e migliorie, con il duplice intento di rendere gli esempi più coerenti e fornire migliori "best practice" per la programmazione in Java, seppure nei limiti che caratterizzano un testo introduttivo. Molti esempi sono stati significativamente riprogettati e reimplementati; altri di scarsa rilevanza sono stati rimossi e ne sono stati aggiunti di nuovi.

I lettori hanno espresso commenti lusinghieri sulle prime tre edizioni di questo manuale, e ciò è stato indubbiamente gratificante per l'autore. Tuttavia vi sono e vi saranno anche alcune critiche: una delle più ricorrenti è relativa alle dimensioni del volume. In proposito, si potrebbe citare il famoso commento dell'Imperatore d'Austria quando, ascoltando un brano suonato da Mozart, esclamò: "Troppe note!". L'autore non ha alcuna pretesa di paragonarsi a Mozart, beninteso, tuttavia è ipotizzabile che chi esprime questo tipo di commento non abbia piena consapevolezza della vastità del linguaggio Java, o che ancora non abbia visto altri manuali sull'argomento.

In ogni caso, in questa edizione si è cercato di eliminare le parti obsolete o quantomeno non essenziali. Di norma è stato controllato tutto, eliminato quanto non era più necessario, apportate le modifiche e si è tentato di migliorare ove possibile. Il materiale eliminato originale è sempre disponibile per il



download dal sito [www.mindview.net](http://www.mindview.net), nelle prime tre edizioni, unitamente ai supplementi a questa edizione.

Ai lettori che ancora non “digeriscono” la corposità di questo manuale vanno le scuse dell'autore, che ha comunque lavorato duramente per contenerne le dimensioni.

## Note sulla grafica di copertina

La copertina di *Thinking in Java* si ispira all'American Arts & Crafts Movement, iniziato alla fine del 1800 e giunto al suo apice tra il 1900 e il 1920. Questo movimento ha avuto origine in Inghilterra come reazione sia alla produzione meccanizzata della Rivoluzione industriale, sia allo stile molto ornamentale dell'epopea Vittoriana. L'Arts & Crafts evidenzia l'essenzialità della progettazione, le forme della natura ereditate dall'Art Nouveau, nonché la manualità e perizia del disegnatore, che ancora non disponeva di strumenti moderni.

Vi sono molte analogie con la situazione odierna: il cambio di secolo, l'evoluzione dalle rozze origini della rivoluzione informatica a qualcosa di più raffinato e ricco di significato, e l'enfasi sulla padronanza del software invece della semplice produzione del codice.

L'autore considera Java nella stessa ottica: un tentativo di elevare il programmatore da semplice meccanico del sistema operativo, per trasformarlo in un artigiano-artista del software.

Sia l'autore sia il disegnatore della copertina, amici fin dall'infanzia, trovano ispirazione in questo movimento, ed entrambi possiedono mobili, lampade e accessori originali o ispirati a questo periodo storico.

L'altro tema della copertina si ispira a un contenitore per la raccolta di insetti, analogo a quelli usati dai naturalisti per preservare i propri esemplari. Questi insetti sono *oggetti* inseriti negli *oggetti-contenitore*, che a loro volta si trovano all'interno dell'*oggetto-copertina*: una metafora che illustra bene il concetto fondamentale di aggregazione nella programmazione a oggetti. Naturalmente un programmatore non potrà esimersi dall'associare gli insetti al termine *bug* (insetto, appunto): i bug che sono stati catturati, presumibilmente resi innocui in un contenitore e infine confinati in un piccolo espositore, quasi a sottolineare la capacità di Java di trovare, visualizzare e neutralizzare i bug. Del resto, questa è effettivamente una delle caratteristiche più potenti del linguaggio.

In questa edizione l'autore ha dipinto anche l'acquerello che è servito da sfondo per l'illustrazione di copertina.



## Ringraziamenti dell'autore

Per prima cosa desidero ringraziare i soci che mi hanno supportato durante i seminari, mi hanno fornito consulenza tecnica e mi hanno aiutato nello sviluppo dei progetti di insegnamento: Dave Bartlett, Bill Vanners, Chuck Allison, Jeremy Meyer e Jamie King. Apprezzo sempre più la vostra pazienza, e apprezzo che personalità indipendenti come le nostre possano collaborare in modo ottimale.

Di recente, senza dubbio grazie a Internet, un gran numero di persone si è associato alle mie imprese, di norma lavorando dai propri uffici. Per collaborare su scala così vasta in passato sarebbe stato necessario prendere in affitto un ambiente piuttosto grande, ma grazie a Internet, alla FedEx e al telefono, mi è possibile usufruire del loro contributo senza costi aggiuntivi.

Nei miei tentativi di imparare a "giocare bene con gli altri", queste persone mi sono state molto utili, e confido di avere l'opportunità di continuare a migliorare il mio lavoro traendo beneficio dagli sforzi altrui. Paula Steuer mi ha fornito un supporto inestimabile mettendo ordine nella gestione della mia attività, e mantenendolo: grazie per avermi stimolato quando era necessario, Paula. Jonathan Wilcox ha passato al setaccio la mia struttura aziendale, rivoltando ogni sasso che potesse nascondere scorpioni o altre insidie, e guidandomi in tutti gli aspetti legali.

Grazie per la vostra attenzione e la vostra tenacia. Sharlynn Cobaugh è ormai diventata un'esperta nell'elaborazione audio e un elemento essenziale nella realizzazione dei corsi multimediali, come pure nella soluzione di altri problemi. Grazie per la perseveranza dimostrata in occasione di problemi informatici all'apparenza insolubili. I collaboratori di Amaio a Praga mi sono stati di aiuto in numerosi progetti. Daniel Will-Harris mi ha ispirato l'idea di lavorare via Internet e naturalmente è stato fondamentale per tutte le progettazioni grafiche. Nel corso degli anni, attraverso conferenze e workshop, Gerald Weinberg è ufficiosamente diventato mio allenatore e mentore, e per questo lo ringrazio.

Ervin Varga si è rivelato straordinariamente efficace per le sue correzioni tecniche sulla quarta edizione: benché altri abbiano fornito aiuto su diversi capitoli ed esempi, Ervin è stato il revisore tecnico principale del manuale, incaricandosi anche di riscrivere la guida alle soluzioni della quarta edizione. Ervin ha corretto molti errori e operato miglioramenti che si sono rivelati inestimabili aggiunte a questo testo. La sua accuratezza e attenzione per i dettagli sono sorprendenti: un grazie a Ervin, certamente il migliore lettore che abbia mai avuto.



Il blog che scrivo sul sito [www.artima.com](http://www.artima.com) di Bill Venners è stato utilissimo per verificare la validità delle mie idee. Ringrazio i lettori che mi hanno aiutato a chiarire molti concetti esprimendo i loro commenti: James Watson, Howard Lovatt, Michael Barker e altri, in particolare quanti mi hanno aiutato con i tipi generici.

Grazie a Mark Gallesi per la sua ininterrotta assistenza.

Evan Cofsky continua a essere un valido sostenitore, grazie alla sua competenza sugli oscuri dettagli di configurazione e gestione dei server web Linux, e per avere ben configurato e reso sicuro il server di MindView.

Un ringraziamento speciale al mio nuovo amico, il caffè, che ha contribuito a generare un immenso entusiasmo per questo progetto. Il bar Camp4 Coffee di Crested Butte (Colorado) è diventato il rifugio principale dei partecipanti ai seminari di MindView, e durante le pause si è dimostrato il miglior fornitore di catering. Un ringraziamento va all'amico Al Smith per averlo creato e reso un luogo così confortevole, e per essere una persona così interessante e divertente. Grazie anche a tutti i baristi del Camp4 Coffee, sempre di ottimo umore.

Desidero ringraziare i collaboratori di Prentice Hall che continuano a soddisfare le mie esigenze, anche quelle più particolari, e per fare del loro meglio per semplificarmi il lavoro.

Alcuni strumenti mi sono stati particolarmente utili durante lo sviluppo, e per questa ragione voglio ringraziare i loro creatori. Cygwin ([www.cygwin.com](http://www.cygwin.com)) ha risolto innumerevoli problemi che Windows non poteva o non voleva risolvere: se soltanto avessi potuto servirmene 15 anni fa, quando ancora utilizzavo Gnu Emacs! Eclipse di IBM ([www.eclipse.org](http://www.eclipse.org)) rappresenta un contributo straordinario alla comunità di sviluppo: mi aspetto grandi cose da questa piattaforma, il cui sviluppo procede ancora oggi. IntelliJ Idea di JetBrains ([www.jetbrains.com](http://www.jetbrains.com)) continua a essere un riferimento creativo tra gli strumenti di sviluppo.

Con questo manuale ho iniziato a usare Enterprise Architect di Sparxsystems ([www.sparxsystems.com](http://www.sparxsystems.com)), che è diventato rapidamente il mio strumento UML preferito. Il formattatore di codice Jalopy di Marco Hunsicker ([www.triemax.com](http://www.triemax.com)) mi è stato utile in numerose occasioni, e Marco si è rivelato impagabile nel configurare questo strumento secondo le mie esigenze. Anche JEdit di Slava Pestov ([www.jedit.org](http://www.jedit.org)) e la sua collezione di plug-in mi sono stati spesso preziosi: JEdit è un eccellente editor per principianti da utilizzare durante i seminari.

Naturalmente, e non lo dirò mai abbastanza, per risolvere i problemi mi serve del linguaggio Python ([www.python.org](http://www.python.org)), delle idee del mio amico Guido



Van Rossum e della banda di geni pazzoidi con i quali ho trascorso giorni memorabili: Tim Peters, ho incorniciato quel mouse che hai preso a prestito, ora ufficialmente chiamato il *TimBotMouse*. Una sola raccomandazione: cercate posti più sani dove pranzare. Grazie ovviamente all'intera comunità di Python, un gruppo di persone straordinarie.

Molte persone mi hanno fatto pervenire le loro correzioni e a tutte sono debitore, ma un grazie particolare (per la prima edizione) va a Kevin Raulerson (scopritore di tonnellate di bug), Bob Resendes (semplicemente incredibile), John Pinto, Joe Dante, Joe Netto (tutti e tre favolosi), David Combs (per i molti chiarimenti e le correzioni di grammatica), Dr. Robert Stephenson, John Cook, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, Steve Wilkinson e un nugolo di altre persone. Il professor Marc Meurrens si è impegnato a fondo per promuovere e realizzare la versione elettronica della prima edizione del manuale disponibile in Europa.

Grazie a quanti mi hanno aiutato a riscrivere gli esempi di utilizzo della libreria Swing (per la seconda edizione) e che mi hanno fornito assistenza: Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajesh Jain, Ravi Manthena, Banu Rajamani, Jens Brandt, Nitin Shivaram, Malcolm Davis, e tutti coloro che mi hanno dato supporto.

Nella quarta edizione, Chris Grindstaff è stato molto utile nello sviluppo della sezione SWT e Sean Neville ha scritto la prima bozza della sezione Flex.

Kraig Brockschmidt e Gen Kiyooka sono stati tra i pochi brillanti tecnici di cui sono diventato amico: autorevoli e fuori dagli schemi, praticano yoga e altre forme di innalzamento spirituale che ritengo veramente ispiratrici ed educative.

Non mi ha sorpreso scoprire quanto la comprensione di Delphi mi sia stata utile per Java, dal momento che questi linguaggi condividono molti concetti e tecniche di progettazione. I miei amici esperti di Delphi hanno contribuito ad approfondire la mia conoscenza di questo meraviglioso ambiente di programmazione. In particolare Marco Cantu (un altro italiano: forse le origini latine comportano un'attitudine per linguaggi di programmazione?), Neil Rubenking (cultore di yoga, cucina vegetariana e Zen finché non ha scoperto i computer) e naturalmente Zack Urlocker (il product manager originale di Delphi), un amico di vecchia data con il quale ho girato il mondo.



Noi tutti siamo debitori dell'intelligenza vivace di Anders Hejlsberg, che continua a lavorare duramente su C# (che, come apprenderete in questo manuale, è stata la fonte principale di ispirazione per Java SE5).

Le intuizioni e il supporto del mio amico Richard Hale Shaw sono stati molto utili, come quelli di Kim, del resto. Richard e io abbiamo trascorso molti mesi tenendo insieme seminari e sforzandoci di fornire ai partecipanti l'esperienza di apprendimento perfetta.

La progettazione del manuale e della copertina, così come l'illustrazione della copertina stessa, sono state curate dal mio amico Daniel Will-Harris, noto autore e designer ([www.will-harris.com](http://www.will-harris.com)), che era solito giocare con lettere adesive mentre aspettava l'invenzione dei computer e del desktop publishing, e che si lamentava di me borbotando sui miei problemi di algebra. In ogni caso, sappiate che io stesso ho prodotto le bozze finali, pertanto eventuali errori sono da imputare soltanto a me.

Ho usato Microsoft Word XP per Windows per scrivere il manuale e creare le pagine già pronte in Adobe Acrobat; il volume è stato realizzato direttamente dai file PDF Acrobat. Quando ho prodotto le versioni finali della prima e della seconda edizione del libro mi trovavo all'estero: come omaggio all'era elettronica, la prima edizione è stata inviata da Città del Capo (Sudafrica) mentre la seconda è stata trasmessa da Praga; la terza e la quarta edizione sono state preparate a Crested Butte, Colorado.

Un ringraziamento speciale va a tutti i miei insegnanti e a tutti i miei studenti (che sono anche i miei insegnanti). Molly, il gatto, si è accoccolato spesso sulle mie ginocchia mentre lavoravo, garantendomi il suo contributo caldo e peloso.

Tra gli amici che mi hanno sostenuto: Patty Gast, Andrew Binstock, Steve Sinosky, JD Hildebrandt, Tom Keffler, Brian McElhinney, Brinkley Barr, Bill Gates del Midnight Engineering Magazine, Larry Constantine e Lucy Lockwood, Gene Wang, Dave Mayer, David Intersimone, Chris e Laura Strand, gli Almquists, Brad Jerbic, Marilyn Cvitanic, Mark Mabry, le famiglie Robbins, le famiglie Moelter e i McMillans, Michael Wilk, Dave Stoner, i Cranstons, Larry Fogg, Mike Sequeira, Gary Entsminger, Kevin e Sonda Donovan, Joe Lordi, Dave e Brenda Bartlett, Patti Gast, Blake, Annette & Jade, i Rentschlers, i Sudeks, Dick, Patty e Lee Eckel, Lynn e Todd e le loro famiglie.

Senza dimenticare, naturalmente, mamma e papà.

# Introduzione

*"E diede all'uomo la parola, e la parola generò il pensiero, che è la misura dell'Universo"*

*Percy Bysshe Shelley, Prometeo liberato (a. II sc. IV)*

*"Gli esseri umani... sono molto spesso alla mercé dello specifico linguaggio che è diventato il mezzo di espressione per la loro società. È illusorio ritenere di poter adattare la realtà in modo fondamentale senza l'uso della lingua, e che la lingua sia soltanto un mezzo fortuito per risolvere gli specifici problemi di comunicazione e riflessione. La questione è che, in larga misura, 'il mondo reale' si è inconsapevolmente sviluppato sulle consuetudini linguistiche del gruppo."*

*Edward Sapir, "La posizione della linguistica come scienza", in Cultura, linguaggio e personalità, Torino. Einaudi, 1972.*

Come qualsiasi linguaggio umano, Java rappresenta un modo per esprimere concetti. Se avrà successo, sarà certamente più semplice e flessibile dei linguaggi alternativi, via via che aumenteranno la dimensione e la complessità dei problemi da risolvere.

Non è possibile considerare Java come una banale raccolta di funzionalità: se considerate singolarmente, alcune di esse non hanno senso. Potete riferirvi alla somma delle parti soltanto pensando in termini di *progettazione*, non di semplice scrittura del codice. E per capire Java in questo modo è necessario comprendere i problemi relativi ai linguaggi e alla programmazione in generale. Questo manuale espone alcuni problemi di programmazione, spiega le ragioni per cui sono definiti tali ed espone le tecniche Java per la loro risoluzione. Quindi, l'insieme di caratteristiche presentato in ciascun capitolo si basa sul modo in cui un determinato tipo di problema viene risolto usando questo linguaggio. Attraverso questo approccio riuscirete, un po' alla volta, a considerare la *forma mentis* Java come vostra lingua madre.

Nel corso del manuale considereremo la necessità di costruire un modello mentale che vi consenta una profonda comprensione del linguaggio; questo vi consentirà, quando incontrerete un problema, di assimilarlo al vostro modello e dedurre la risposta.



## Prerequisiti

Questo manuale presuppone una certa pratica di programmazione: occorre sapere che un programma è un insieme di dichiarazioni; comprendere i concetti di *routine*, funzione, macro; conoscere le istruzioni di controllo del flusso programmatico, come *if*, i costrutti per la gestione dei cicli, come *while*, e così via. Potreste aver appreso queste nozioni lavorando in ambienti diversi, per esempio programmando con un macrolinguaggio oppure utilizzando uno strumento come Perl: in ogni caso, purché abbiate sufficiente esperienza da sentirvi a vostro agio con i concetti fondamentali della programmazione, potrete trarre vantaggio dalla lettura di questo volume.

Naturalmente sarà più facile seguire questo manuale per i programmatore C, e ancora di più per i programmatore C++. Non preoccupatevi, tuttavia, se non siete esperti in questi linguaggi: che lo siate o meno, quello che serve è tanta buona volontà e duro lavoro. Inoltre, il seminario multimediale *Thinking in C*, che potete scaricare da [www.mindview.net](http://www.mindview.net), vi fornirà i fondamenti di programmazione necessari per apprendere Java. Comunque sia, vedrete i concetti della programmazione a oggetti (OOP) e i meccanismi di controllo di base Java.

Malgrado la presenza di rimandi a funzionalità tipiche di C e C++, questi non devono essere visti come considerazioni riservate agli eletti, bensì un supporto affinché come programmatore possiate mettere Java in relazione con questi linguaggi, dai quali, dopotutto, è derivato. L'autore ha avuto cura di semplificare al massimo questi riferimenti, chiarendo eventuali argomenti che possano risultare poco familiari a un programmatore non esperto di C/C++.

## Imparare Java

Nello stesso periodo in cui è stato pubblicato il suo primo libro, *Using C++* (Osborne/McGraw-Hill, 1989), l'autore ha iniziato a studiare Java. Insegnare idee e concetti di programmazione è poi diventata la sua professione: in tutto il mondo, fin dal 1987 l'autore ha visto persone assentire col capo, sguardi vuoti ed espressioni perplesse tra il pubblico.

Dopo aver iniziato l'istruzione aziendale di gruppi più piccoli, durante le esercitazioni l'autore si è reso conto che anche quanti sorridevano e chinavano il capo in segno di assenso erano poi confusi su molti argomenti. Organizzando e presiedendo per molti anni i corsi di C++ alla Software Development Conference, e in seguito quelli di Java, l'autore ha scoperto che



i relatori avevano la tendenza a fornire al pubblico troppi argomenti e più velocemente del dovuto.

Era evidente che in tal modo, sia per il diverso livello dei partecipanti sia per il modo in cui venivano presentati gli argomenti, si sarebbe finito per perdere l'attenzione di una parte del pubblico. Forse perché refrattario ai tradizionali metodi di insegnamento (avversione che per molte persone nasce dalla noia), l'autore ha ritenuto di dare una svolta realizzando presentazioni di tipo diverso, tutte piuttosto brevi, ricorrendo all'esperienza acquisita dalla sperimentazione e dalla ripetizione, una tecnica che peraltro funziona bene anche nella progettazione software.

Alla fine, utilizzando quanto appreso dalla propria esperienza di insegnamento, l'autore ha sviluppato un tipo di corso che la sua società, MindView Inc., offre come seminario intitolato *Thinking in Java*, tenuto sia internamente sia per il grande pubblico: è il principale corso introduttivo di MindView, atto a fornire le basi necessarie per corsi più avanzati. Sul sito [www.mindview.net](http://www.mindview.net) potete trovare maggiori informazioni su questo seminario, disponibile anche nel CD-ROM *Hands-On Java*.

Il feedback ottenuto da ogni corso permette di modificare e riorganizzare la materia, fino a trovare la tipologia di insegnamento più adatta. Questo libro, tuttavia, non è una semplice raccolta di annotazioni per un corso, ma cerca di fornire quante più informazioni possibile, strutturando i diversi temi per guidarvi all'argomento successivo. Più di ogni altra cosa, il manuale è stato concepito per aiutare il lettore ad affrontare in modo autonomo un nuovo linguaggio di programmazione.

## Obiettivi

Come il titolo precedente, *Thinking in C++*, questo libro è stato progettato con un obiettivo preciso: fornire gli elementi per apprendere un linguaggio di programmazione. Quando progetta un capitolo del manuale, l'autore tiene conto anche del risultato di una buona lezione tenuta nei suoi corsi. Il feedback del pubblico ha consentito di individuare i concetti più complessi, che richiedono maggiore approfondimento. Laddove l'ambizione aveva portato a esaminare un numero eccessivo di funzionalità, durante i corsi si è poi compreso che questo creava confusione negli studenti.

Gli obiettivi principali di questo manuale sono descritti di seguito.

1. Presentare il materiale un passo alla volta, per consentire di assimilare facilmente ogni concetto prima di passare a un nuovo argomento. La presentazione delle varie caratteristiche è stata ordinata accuratamente,

in modo che ciascun argomento sia esposto nella sua interezza prima di vederlo in azione. Naturalmente questo non è sempre possibile: in questi casi, è fornita una breve descrizione introduttiva.

2. Ricorrere a esempi quanto più semplici e brevi possibile. Questo approccio talvolta impedisce di affrontare problemi derivati dal mondo reale; tuttavia, il principiante trova più gratificante riuscire a capire ogni dettaglio di un esempio, che non essere impressionato dall'ampiezza del problema che viene risolto. Bisogna anche tenere conto di un serio limite alla quantità di codice che può essere assimilato in aula. Per questa ragione l'autore si aspetta di ricevere critiche per i suoi "esempi giocattolo", ma è pronto ad affrontarle per realizzare un prodotto utile anche dal punto di vista pedagogico.
3. Formire le competenze importanti ai fini della comprensione del linguaggio, anziché tutto il nozionismo dell'autore. Esiste una gerarchia nell'importanza delle informazioni: vi sono alcune nozioni che il 95% dei programmati non avrà occasione di usare, dettagli che confondono lo studente e aumentano la sua percezione della complessità del linguaggio. Considerate un semplice esempio preso dal linguaggio C nel quale, memorizzando la tavola delle precedenze degli operatori (operazione che l'autore non ha mai fatto), è possibile scrivere dell'ottimo codice. Se si tiene conto anche di questi argomenti, il semplice lettore o manutentore del codice risulterà confuso: è quindi preferibile ignorare la precedenza degli operatori e servirsi delle parentesi in caso di dubbi.
4. Mettere a fuoco ogni sezione, in modo che il tempo richiesto per la lettura e quello necessario per eseguire le esercitazioni siano contenuti. Questo non solo fa sì che la mente del pubblico sia più attenta e coinvolta durante un workshop, ma fornisce al lettore una più intensa sensazione di autorealizzazione.
5. Fornire una base solida che consenta di assimilare gli argomenti in modo adeguato prima di passare a corsi o manuali più approfonditi.

## Imparare da questo libro

L'edizione originale di questo manuale è stata sviluppata traendo spunto da un corso di una settimana: un periodo di tempo sufficiente, quando Java era nella sua infanzia, per illustrare le caratteristiche del linguaggio. Via via che Java cresceva e si arricchiva di funzionalità e librerie, l'autore ha tentato ostinatamente di contenere tutto in un periodo di tempo così breve, fino a quando un cliente non chiese che venissero spiegate "soltanto le basi". A



quel punto, l'autore si è reso conto che il tentativo di insegnare tutto in una settimana era diventato angosciante sia per il docente sia per i partecipanti: Java non era più il linguaggio "semplice" che potesse essere spiegato compiutamente in pochi giorni.

Questa esperienza ha governato gran parte della riorganizzazione di questo libro, ora progettato per supportare un seminario della durata di due settimane oppure due trimestri di corso universitario. La parte introduttiva termina con il Capitolo 1 del Volume 2, "Gestione degli errori con le eccezioni", e può essere integrata con un'introduzione a JDBC, Servlet e JSP; questa sezione rappresenta un corso di base, ed è il nucleo del CD-ROM *Hands-On Java*. Le altre parti del manuale formano un corso di livello intermedio, corrispondente al materiale presente nel CD-ROM *Intermediate Thinking in Java*. Entrambi i CD-ROM sono in vendita su [www.mindview.net](http://www.mindview.net); per informazioni sui sussidi ai docenti relativi a questo manuale potete contattare l'editore Pearson Education Italia all'indirizzo <http://lhp.e.pearsoned.it>.

## Documentazione JDK in HTML

Il linguaggio e le librerie Java di Sun Microsystems, fornite gratuitamente all'indirizzo <http://java.sun.com>, vengono resi disponibili insieme con documentazione in forma elettronica, che potete consultare con qualsiasi browser web. Molti manuali su Java hanno duplicato questa documentazione: pertanto, dal momento che potrete già disporre di questa documentazione, e in ogni caso non avrete difficoltà a procurarvela in qualsiasi momento, questo manuale non ne ripeterà il contenuto. Del resto, non solo è più rapido trovare la descrizione di una classe cercandola con il browser web, rispetto alla consultazione di un libro, ma probabilmente la documentazione in linea è più aggiornata: troverete dunque alcuni rimandi alla "documentazione JDK". Vi saranno fornite descrizioni supplementari delle classi soltanto quando sia necessario integrare la documentazione per consentirvi di comprendere a fondo qualche specifico esempio.

## Esercizi

Si è notato che le esercitazioni sono incredibilmente utili al fine di perfezionare la comprensione di uno studente; per questo motivo, al termine di ogni capitolo vengono presentati alcuni esercizi. La maggior parte è ideata per essere svolta in un periodo di tempo ragionevole in aula sotto la supervisione di un docente, per assicurarsi che tutti gli studenti abbiano recepito corret-



tamente le nozioni fornite: alcuni esercizi sono più stimolanti, in ogni caso nessuno di essi rappresenta una sfida particolare.

Potete trovare le soluzioni di tali esercizi nel documento elettronico *The Thinking in Java Annotated Solution Guide*, in vendita su [www.mindview.net](http://www.mindview.net).

## Fondamenti di Java

Un altro vantaggio di questa edizione è il seminario multimediale gratuito *Thinking in C*, scaricabile da [www.mindview.net](http://www.mindview.net), che vi fornirà un'introduzione alla sintassi, agli operatori e alle funzioni del linguaggio C su cui si basa la sintassi Java. Nelle edizioni precedenti era presente nel CD-ROM *Foundations for Java* allegato al volume, ora potete scaricarlo gratuitamente.

La tecnologia si è evoluta, pertanto *Thinking in C* è stato rielaborato come presentazione Flash scaricabile da Internet. Mettendo a disposizione questo seminario in linea, invece che in un CD-ROM allegato, l'autore si assicura che ognuno possa partire con una preparazione adeguata.

Inoltre la presenza di questo seminario in linea consente l'accesso a questo libro da parte di un pubblico più vasto. Sebbene i Capitoli 3 e 4 del Volume 1, "Gli operatori" e "Il controllo dell'esecuzione" trattino gli elementi fondamentali di Java che discendono dal C, il corso in linea offre un'introduzione più semplice e libera dai presupposti sulle competenze di programmazione dello studente, a differenza del manuale.

## Codice sorgente

Tutto il codice sorgente presente in questo libro è *copyrighted freeware*, vale a dire disponibile gratuitamente ma soggetto a diritti d'autore, ottenibile come singolo pacchetto sul sito [www.mindview.net](http://www.mindview.net). Per assicurarvi di disporre della versione più recente, ricordate che questo sito è il distributore ufficiale del codice. Siete liberi di utilizzare il codice in aula e in ambiente didattico.

L'obiettivo principale dei diritti d'autore è fare in modo che la fonte del codice venga segnalata correttamente e che tale codice non possa essere ripubblicato senza autorizzazione. Se la fonte del codice è citata, l'uso degli esempi nella maggior parte dei media è generalmente consentito. In ogni file sorgente troverete quindi un riferimento alla seguente informativa:

```
///! Copyright.txt
```

```
This computer source code is Copyright ©2006 MindView, Inc.
```



All Rights Reserved.

Permission to use, copy, modify, and distribute this computer source code (Source Code) and its documentation without fee and without a written agreement for the purposes set forth below is hereby granted, provided that the above copyright notice, this paragraph and the following five numbered paragraphs appear in all copies.

1. Permission is granted to compile the Source Code and to include the compiled code, in executable format only, in personal and commercial software programs.
2. Permission is granted to use the Source Code without modification in classroom situations, including in presentation materials, provided that the book "Thinking in Java" is cited as the origin.
3. Permission to incorporate the Source Code into printed media may be obtained by contacting:

MindView, Inc. 5343 Valle Vista La Mesa, California 91941  
[Wayne@MindView.net](mailto:Wayne@MindView.net)

4. The Source Code and documentation are copyrighted by MindView, Inc. The Source code is provided without express or implied warranty of any kind, including any implied warranty of merchantability, fitness for a particular purpose or non-infringement. MindView, Inc. does not warrant that the operation of any program that includes the Source Code will be uninterrupted or error-free. MindView, Inc. makes no representation about the suitability of the Source Code or of any software that includes the Source Code for any purpose. The entire risk as to the quality and performance of any program that includes the Source Code is with the user of the Source Code. The user



understands that the Source Code was developed for research and instructional purposes and is advised not to rely exclusively for any reason on the Source Code or any program that includes the Source Code. Should the Source Code or any resulting software prove defective, the user assumes the cost of all necessary servicing, repair, or correction.

5. IN NO EVENT SHALL MINDVIEW, INC., OR ITS PUBLISHER BE LIABLE TO ANY PARTY UNDER ANY LEGAL THEORY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS, OR FOR PERSONAL INJURIES, ARISING OUT OF THE USE OF THIS SOURCE CODE AND ITS DOCUMENTATION, OR ARISING OUT OF THE INABILITY TO USE ANY RESULTING PROGRAM, EVEN IF MINDVIEW, INC., OR ITS PUBLISHER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. MINDVIEW, INC. SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOURCE CODE AND DOCUMENTATION PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, WITHOUT ANY ACCOMPANYING SERVICES FROM MINDVIEW, INC., AND MINDVIEW, INC. HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Please note that MindView, Inc. maintains a Web site which is the sole distribution point for electronic copies of the Source Code, <http://www.MindView.net> (and official mirror sites), where it is freely available under the terms stated above.

If you think you've found an error in the Source Code, please submit a correction using the feedback system that you will find at <http://www.MindView.net>.

///:~



Potete usare liberamente il codice nei vostri progetti, in aula e nelle vostre presentazioni, purché sia presente l'informativa sui diritti d'autore che appare in ogni file sorgente.

### **Standard di codifica**

Nel testo di questo manuale, gli identificatori (metodi, variabili e nomi di classe) sono evidenziati in **grassetto**. Anche la maggior parte delle parole chiave è riportata in grassetto, tranne quelle che sono usate così spesso da rendere il grassetto ridondante, per esempio "class".

Negli esempi di questo libro si è fatto ricorso a uno stile di codifica particolare, per quanto possibile conforme a quello utilizzato da Sun in quasi tutti i frammenti di codice che trovate sul suo sito web (<http://java.sun.com/docs/codestyle/index.html>): uno stile che sembra essere supportato dalla maggior parte degli ambienti di sviluppo Java.

Se avete letto nella versione originale altri manuali scritti dall'autore, avrete certamente notato che il suo stile di codifica coincide con quello di Sun. Questo è gratificante sebbene l'autore, per quanto a sua conoscenza, non abbia nulla a che fare con questa affinità.

Lo stile di formattazione è un argomento che potrebbe essere oggetto di ore di discussione; per questa ragione l'autore non ha la pretesa di imporre la propria come forma corretta, basandosi sugli esempi forniti: l'autore ha motivazioni personali per ricorrere a questo stile. Poiché Java è un linguaggio di programmazione nel quale lo stile non è vincolante, potete continuare a usare quello con cui vi trovate a vostro agio. Un modo per ovviare al problema della codifica consiste nell'uso di uno strumento come *Jalopy* ([www.triemax.com](http://www.triemax.com)), rivelatosi prezioso nello sviluppo di questo manuale, con cui modificare la formattazione nel modo che preferite.

I file sorgenti presenti nel libro sono stati testati con un sistema automatico e dovrebbero funzionare tutti senza errori di compilazione. Come si è detto, questo manuale tratta di Java SE5/6: qualora abbiate necessità di esaminare argomenti relativi alle altre versioni del linguaggio, ricordate che le precedenti edizioni sono liberamente scaricabili presso [www.mindview.net](http://www.mindview.net).

### **Errori**

Per quanti accorgimenti uno scrittore adotti per rilevare gli errori, alcuni di essi si insinuano sempre nel suo lavoro, spesso particolarmente evidenti a un lettore "fresco". Se notate qualsiasi cosa che ritenete sia un errore tecnico



o del codice, utilizzate il link a questo manuale sul sito [www.mindview.net](http://www.mindview.net), indicando l'errore e la correzione che intendete proporre; per segnalare refusi o errori di traduzione relativi alla traduzione italiana contattate Pearson Education Italia all'indirizzo di posta elettronica [editoriale@pearson.com](mailto:editoriale@pearson.com). La vostra collaborazione sarà apprezzata.

# Capitolo 1

## La concorrenza



Fino a questo punto del volume è stata trattata la *programmazione sequenziale*, concetto secondo il quale, in un programma, tutto avviene un passo per volta.

Un numero elevato di problemi di programmazione può essere risolto utilizzando la programmazione sequenziale. Tuttavia, in alcuni casi diventa pratico, se non addirittura essenziale, eseguire parallelamente più parti di un programma, facendo in modo che tali parti sembrino eseguite nello stesso tempo; è ovvio che se il computer dispone di processori multipli non si tratterà soltanto di una finzione.

La programmazione parallela o concorrente può produrre miglioramenti rilevanti in termini di velocità di esecuzione dei programmi, semplificare il modello di progettazione per determinati tipi di programmi o offrire entrambi i vantaggi. Comunque sia, diventare esperti della teoria e delle tecniche di programmazione concorrente non è facile, poiché si tratta di un argomento piuttosto avanzato. Questo capitolo deve essere visto come una semplice introduzione sul tema.

Come vedrete, il vero problema con la programmazione concorrente si verifica quando le operazioni che stanno funzionando in parallelo iniziano a interferire tra loro. Questo significa che sebbene sia possibile realizzare programmi simultanei che, con la necessaria attenzione e un accurato controllo del codice, funzionano correttamente, nella pratica, però, è molto più facile scrivere programmi concorrenti che "sembrano" funzionare, ma che in presenza di determinate condizioni



daranno problemi. Queste condizioni possono non verificarsi mai o avvenire così di rado da non essere avvertite in fase di test. In effetti potreste non essere in grado di scrivere test che generino le condizioni di errore adatte per il vostro programma concorrente. Spesso i problemi si presenteranno soltanto in situazioni particolari, quasi certamente sotto forma di reclami da parte del cliente. Questo è uno degli argomenti più complessi nello studio della programmazione concorrente: se lo ignorate, ne sarete probabilmente travolti.

Da quanto detto la programmazione sembra molto "pericolosa"; il fatto che possiate avvertire un po' di timore non è detto che sia un atteggiamento negativo. Anche se Java SE5 ha apportato notevoli miglioramenti alla programmazione concorrente, non sono ancora disponibili "reti di sicurezza" come la verifica in fase di compilazione o le eccezioni controllate, che possono segnalarti eventuali errori. Nella programmazione concorrente siete soli con voi stessi; soltanto un approccio sospettoso e contemporaneamente aggressivo vi consentirà di scrivere codice Java multithread affidabile.

Talvolta viene fatto notare che la programmazione concorrente è un argomento troppo avanzato per essere incluso in un testo introduttivo a un linguaggio di programmazione. Molti sostengono anche che la concorrenza è un argomento a sé stante che può essere trattato indipendentemente dal linguaggio, e che i pochi casi in cui compare nella programmazione quotidiana, per esempio nelle interfacce grafiche, possono essere gestiti ricorrendo a forme idiomatiche speciali. Allora, perché introdurre un argomento tanto complesso se è possibile evitarlo?

La situazione, purtroppo, non è così semplice: non potete scegliere il momento in cui i thread compariranno nei vostri programmi Java; soltanto per il fatto che non aviate mai i thread direttamente non è detto che possiate evitare di includerli nel vostro codice. Per esempio, le applicazioni web sono tra le applicazioni Java più comuni, e la classe di base della libreria web, il *servlet*, è intrinsecamente multithread: una condizione essenziale, poiché i server web spesso sono sistemi multiprocessore e la concorrenza è un meccanismo ideale per utilizzare questi sistemi. Per quanto semplice possa sembrare un servlet, dovete comprendere i problemi della programmazione concorrente per utilizzarlo correttamente; come vedrete nel Capitolo 2, questa considerazione vale anche per le interfacce grafiche. Anche se entrambe le librerie SWT e Swing hanno funzionalità per la sicurezza dei thread, è difficile capire come servirsene nel modo migliore senza avere prima compreso i meccanismi della programmazione concorrente.

Java è un linguaggio multithread e i problemi relativi alla concorrenza esistono, anche se non ne siete consapevoli. Questo è uno dei motivi per cui oggi sono in circolazione numerosi programmi Java che funzionano solo per caso, o che



funzionano "quasi" sempre, salvo misteriosamente bloccarsi ogni tanto a causa di problemi di concorrenza non rilevati. Talvolta queste interruzioni operative sono benigne, altre volte provocano la perdita di dati importanti, e se non foste al corrente dei problemi connessi con la programmazione concorrente potreste concludere che il problema non sia nel software ma altrove. Si tratta di problemi che possono anche diventare evidenti o addirittura amplificarsi, se un programma viene spostato su un sistema multiprocessore. Sostanzialmente, la conoscenza della programmazione concorrente vi aiuta anche a comprendere che programmi all'apparenza corretti possono presentare comportamenti errati.

Apprendere la programmazione concorrente equivale a entrare in un nuovo mondo e imparare un nuovo linguaggio, o quantomeno un nuovo insieme di concetti del linguaggio. L'autore ritiene che la comprensione della programmazione concorrente sia di difficoltà paragonabile alla comprensione dei concetti OOP. Impegnandovi a fondo riuscirete ad afferrare il meccanismo di base, ma di solito sono necessari uno studio e una comprensione veramente profondi per carpirne tutti i segreti.

## Molteplici aspetti della programmazione concorrente

Uno dei motivi che generano maggiore confusione nella programmazione concorrente è il fatto che l'utilizzo della concorrenza implica non soltanto la risoluzione di diversi problemi, ma anche la scelta tra una varietà di approcci implementativi e nessuna corrispondenza lineare tra i due aspetti; come se non bastasse, talvolta anche i contorni dei problemi non sono ben delineati. Di conseguenza, prima di riuscire a servirvi in modo efficace delle tecniche di programmazione concorrente dovete assolutamente capire tutti i problemi e i casi particolari. I problemi risolvibili con la programmazione concorrente possono essere classificati, con una certa approssimazione, nelle categorie "velocità" e "maneggevolezza della progettazione".

### ***Velocità di esecuzione***

All'apparenza il problema della velocità è semplice: se volete che l'esecuzione di un programma sia più rapida, dividetelo in porzioni ed eseguite ogni parte su un processore separato. La concorrenza è uno strumento fondamentale per la programmazione su sistemi multiprocessore. Con gli effetti della cosiddetta legge di Moore ([http://it.wikipedia.org/wikil/Legge\\_di\\_Moore](http://it.wikipedia.org/wikil/Legge_di_Moore)) ormai agli sgoccioli, almeno per quanto riguarda i circuiti integrati convenzionali, i miglioramenti di velocità si conseguono soprattutto grazie a processori *multicore* invece che per



mezzo di circuiti integrati più veloci. Per eseguire più velocemente i programmi dovete imparare a trarre vantaggio da questi processori supplementari: questo è uno dei benefici che la programmazione concorrente può offrirvi.

Su un sistema multiprocessore potete distribuire più operazioni su diverse CPU, ottenendo miglioramenti enormi in termini di rendimento globale. Pensate ai server web multiprocessore, in grado di distribuire le moltissime richieste degli utenti sulle varie CPU disponibili, grazie a programmi che assegnano un thread per ogni richiesta.

Spesso, però, la programmazione concorrente può anche migliorare le prestazioni di programmi che lavorano su un solo processore.

Un controsenso, all'apparenza: infatti, potreste ritenere che un programma multithread che lavora su un singolo processore debba avere un sovraccarico operativo maggiore rispetto a un analogo programma in cui tutti i componenti operano in modo sequenziale, dovuto all'onere aggiuntivo del cosiddetto "cambio di contesto" (*context switch*), che si incarica di passare da un'operazione a un'altra. A una valutazione superficiale sembrerebbe meno oneroso eseguire tutte le parti di un programma come un'unica operazione, risparmiando il costo aggiuntivo del cambio di contesto.

Ciò che può fare la differenza è il *bloccaggio*. Se un'operazione o task del programma non può continuare a causa di una condizione che esula dal controllo del programma stesso (di norma, un problema di I/O), si è soliti dire che il task o il thread si bloccano. Se non avete optato per la programmazione concorrente, l'intera applicazione si interromperà fino a quando la condizione esterna non subirà modifiche. Invece, se il programma è stato scritto in un'ottica di concorrenza, le altre operazioni del programma potranno continuare mentre il task è bloccato, e l'applicazione continuerà con le attività previste. Infatti, dal punto di vista delle prestazioni, non ha senso utilizzare la concorrenza su un computer monoprocessoresso, a meno che uno dei task non sia a rischio di blocco.

Un esempio molto comune del miglioramento di prestazioni ottenibile nei sistemi monoprocessoresso è la *programmazione a eventi* (*event-driven*). Uno dei motivi principali per utilizzare la concorrenza, in effetti, è produrre un'interfaccia utente reattiva. Considerate un programma che esegue una qualsiasi operazione di lunga durata, pertanto senza input da parte dell'utente e senza alcuna reazione. Se avete previsto un tasto "Interrompi" non vorrete certo essere costretti a controllare la condizione "premuto/non premuto" in ogni frammento di codice che scrivete! Questo approccio produrrebbe codice terribile, sempre ammesso che il programmatore non dimentichi di eseguire questi controlli. Senza la concorrenza, l'unico modo per realizzare un'interfaccia utente reattiva è fare in modo che tutti i task eseguano periodicamente controlli per verificare l'eventuale input da parte dell'utente. Creando un



thread di esecuzione separato per rispondere all'input dell'utente, anche se questo thread sarà bloccato per la maggior parte del tempo il programma garantirà un certo livello di disponibilità.

Il programma deve continuare a eseguire le sue operazioni e nello stesso tempo deve restituire il controllo all'interfaccia, in modo da dialogare con l'utente. Ma un metodo convenzionale può comportarsi in questo modo? In effetti questo sembrerebbe impossibile, sarebbe come se la CPU dovesse trovarsi in due posti: ma è precisamente l'"illusione" prodotta dalla concorrenza e, nel caso dei sistemi multiprocessore, è più di un'illusione.

Una tecnica molto diretta per implementare la concorrenza è quella che agisce a livello del sistema operativo, che ricorre ai *processi*; un processo è un programma autonomo che funziona all'interno del proprio spazio di memoria. Un sistema operativo *multitasking* riesce a gestire più processi (programmi) per volta commutando la CPU da un processo all'altro, dando così l'impressione che ogni processo funzioni in modo autonomo.

Quella dei processi è una tecnica molto interessante, poiché di norma il sistema operativo isola i singoli processi per evitare che interferiscano l'uno con l'altro, semplificando in un certo senso la programmazione. Di contro i sistemi concorrenti come quello utilizzato in Java condividono le risorse, quali la memoria e i dispositivi di input/output; pertanto la difficoltà principale nella scrittura di programmi multithread è coordinare l'utilizzo di queste risorse tra i vari task associati ai thread, in modo che non possano essere utilizzate da più di un task per volta.

Considerate questo semplice esempio che utilizza i processi del sistema operativo. L'autore, mentre scrive un libro, esegue regolarmente diverse copie di backup del lavoro corrente: una su disco, una sulla chiavetta USB, una su disco zip e un'altra su un sito ftp remoto. Per automatizzare questo processo l'autore ha realizzato un piccolo programma in Python (ma i concetti sono indipendenti dal linguaggio) che archivia il libro in formato zip, assegnando al file un nome completo del numero di versione: a quel punto effettua le copie di backup necessarie.

Inizialmente l'autore ha scelto di eseguire tutte le copie in sequenza, aspettando il completamento di ciascuna prima di iniziare quella seguente; in seguito, però, si è reso conto che ogni operazione di copia richiedeva tempi diversi in funzione della velocità di I/O del supporto. Poiché utilizzava un sistema operativo multitasking, l'autore ha poi scelto di avviare ogni operazione di copia come processo separato, facendo in modo che tutti venissero eseguiti parallelamente. Questa riscrittura ha di fatto accelerato l'esecuzione dell'intero programma: mentre un processo è bloccato, l'altro può continuare in modo indipendente.



Questo è un esempio ideale di concorrenza: ogni operazione viene eseguita come processo nel suo spazio di memoria, di conseguenza non esiste rischio di interferenza tra i task. Ma soprattutto non è necessario che i task comunichino a vicenda, poiché sono indipendenti. È il sistema operativo a occuparsi di tutti i dettagli per garantire che le operazioni di copia dei file avvengano in modo appropriato: questo, ovviamente, si traduce in assenza di rischi e maggiore velocità del programma, in pratica “a costo zero”.

C’è chi sostiene che i processi siano l’unico approccio ragionevole alla concorrenza, purtroppo, però, la quantità e il tipo di limitazioni dei processi sono tali da impedirne l’applicazione in tutte le situazioni in cui essa è coinvolta.<sup>1</sup>

Alcuni linguaggi di programmazione sono stati progettati per mantenere isolati i task concorrenti: si tratta dei cosiddetti *linguaggi funzionali*, nei quali ogni chiamata di funzione non produce effetti secondari (per non interferire con altre funzioni) e può quindi essere condotta come operazione indipendente. *Erlang* ([www.erlang.org](http://www.erlang.org)) è uno di questi linguaggi, che include meccanismi di sicurezza per consentire la comunicazione tra i task.

Se ritenete che una parte del vostro programma debba fare ricorso pesantemente alla concorrenza, e se pensate che lo sviluppo di tale porzione di codice sia troppo problematica, potreste prendere in considerazione la possibilità di realizzare quella parte di programma direttamente in un linguaggio dedicato alla concorrenza, quale Erlang.

Java ha adottato un approccio più tradizionale, integrando il supporto ai thread in un linguaggio sequenziale.<sup>2</sup>

Anziché ricorrere al *forking*<sup>3</sup> di processi esterni in un sistema operativo multitasking, in Java l’utilizzo dei thread (*threading*) crea i task all’interno del singolo processo rappresentato dal programma in esecuzione. Un vantaggio fornito da questa tecnica è stata la trasparenza nei confronti del sistema operativo, un obiettivo progettuale importante per Java.

Per esempio, le versioni del sistema operativo Apple precedenti Mac OS X, che costituivano un obiettivo determinante per le prime release di Java, non supportavano l’elaborazione multitasking; finché Java non ha iniziato a of-

---

1. Eric Raymond, per esempio, ne fa un caso esemplare in *The Art of UNIX Programming* (Addison-Wesley, 2004).

2. Alcuni sostengono che il tentativo di abbinare la concorrenza a un linguaggio sequenziale sia un approccio sbagliato, tuttavia spetta a voi trarre le debite conclusioni.

3. Una fork in informatica è la modalità attraverso cui un processo informatico crea una copia di se stesso, che si comporta come un “processo figlio” nei confronti del processo originale che viene chiamato “processo padre”. Più in generale, una fork in un ambiente multithreading significa che un thread in esecuzione è stato duplicato.



frire il supporto al multithreading, nessun programma concorrente Java era portabile sulle piattaforme Macintosh, annullando, di fatto, la regola del "scritto una volta/funzionante ovunque".

### ***Migliorare la progettazione del codice***

Un programma che utilizza task multipli su un computer monoprocessoressegue soltanto un'attività per volta, pertanto teoricamente sarebbe possibile scrivere lo stesso programma senza neppure ricorrere ai task. Tuttavia, la concorrenza offre un vantaggio notevole in termini di organizzazione, semplificando la progettazione dei programmi. Alcuni tipi di problemi, quali la simulazione, sono difficili da risolvere senza il supporto alla concorrenza.

La maggior parte dei lettori ha visto almeno una forma di simulazione, per esempio videogiochi o animazioni generate da computer e inserite in un film. Di solito le simulazioni coinvolgono l'interazione di molti elementi, ciascuno con una propria "mente". Sebbene possiate ribadire che, su un sistema monoprocessoress, ogni elemento della simulazione viene portato avanti dalla stessa CPU, dal punto di vista della programmazione è molto più facile fingere che ogni elemento abbia un suo processore e sia un'operazione indipendente.

Una simulazione completa può implicare un numero enorme di task, in funzione del fatto che ogni suo elemento può comportarsi in modo indipendente: questo include anche portoni e rocce, non soltanto folletti e maghi. Spesso i sistemi multithread impongono un limite relativamente ridotto al numero di thread disponibili, a volte dell'ordine di decine o centinaia; il numero esatto varia in base a condizioni fuori dal controllo del programma: può dipendere dalla piattaforma, oppure, nel caso di Java, dalla versione della JVM. In Java potete dare sempre per scontato che non avrete abbastanza thread disponibili per assegnarne uno a ogni elemento di una simulazione lunga o complessa.

Un approccio tipico per risolvere il problema è il ricorso al cosiddetto multithreading *cooperativo*. Il threading Java, invece, è di tipo *preemptive* (letteralmente, con prelazione): questo significa che un meccanismo di pianificazione assegna tempistiche a ogni thread, interrompendo periodicamente un thread e commutando il contesto a un altro thread, in modo che a ciascuno sia attribuito un tempo sufficiente per pilotare la propria operazione. In un sistema cooperativo ogni operazione cede volontariamente tale controllo, e questo richiede che il programmatore inserisca un'istruzione specifica per rendere produttiva ciascuna operazione. Il vantaggio di un sistema cooperativo è duplice: il cambio di contesto è normalmente meno oneroso rispetto a un sistema preemptive e in teoria non vi è limite al numero di task indipendenti che possono funzionare simultaneamente. Quando dovete gestire numerosi elementi di simulazione questa può essere la soluzione ideale: tenete presente, tuttavia,



che alcuni sistemi cooperativi non sono stati progettati per distribuire i task tra i diversi processori, e questa può essere una limitazione notevole.

All'estremo opposto, la concorrenza è un modello molto utile per lavorare con i moderni sistemi di *messaggistica*, che funzionano effettivamente in questo modo, coinvolgendo numerosi computer indipendenti distribuiti su una rete. In questo caso tutti i processi sono attivi in modo autonomo e non vi è alcuna possibilità che le risorse vengano condivise. È tuttavia necessario sincronizzare il trasferimento di informazioni tra i processi, in modo che l'intero sistema di messaggistica non smarrisca informazioni o non ne accetti in tempi errati. Anche se non prevedete di servirvi della concorrenza in futuro è utile che la comprendiate a fondo per afferrare i meccanismi architetturali della messaggistica, una tecnica che sta diventando predominante nella creazione di sistemi distribuiti.

La concorrenza impone costi, inclusi quelli legati alla sua complessità, che però di norma sono equilibrati da miglioramenti nel progetto del programma, dal bilanciamento delle risorse e dalla praticità di utilizzo offerta all'utente. In generale i thread consentono di realizzare un progetto meno vincolante, in alternativa al quale una parte del codice dovrebbe essere dedicata esplicitamente al monitoraggio di attività che di solito sono gestite dai thread.

## Gestione di base dei thread

Grazie alla programmazione concorrente potete suddividere un programma in attività separate che funzionano in modo indipendente. Nel multithreading, ciascuno di questi task autonomi, chiamati anche *sottotask* o *subtask*, è pilotato da un *thread di esecuzione* (*thread of execution*). Un *thread* è un singolo flusso di controllo sequenziale all'interno di un processo: pertanto un unico processo può avere diversi task in esecuzione simultanea, ma per il programmatore è come se ogni task avesse una CPU a propria disposizione. È un meccanismo secondario che si occupa di distribuire il tempo di CPU, ma generalmente non dovrete preoccuparvene.

In sintesi, il modello di threading è una tecnica di programmazione di comodo, che facilita la manipolazione contestuale di diverse operazioni nell'ambito di uno stesso programma: la CPU passerà da una all'altra, assegnando a ogni operazione il tempo di esecuzione necessario.<sup>4</sup>

---

4. Questo è vero se il sistema operativo offre la funzionalità di *time slicing* (suddivisione dei tempi), come è il caso, per esempio, di Windows. Solaris applica un modello di concorrenza FIFO: a meno che non si attivi un thread ad alta priorità, quello corrente continua a funzionare finché non si blocca o termina. Questo significa che altri thread con la stessa priorità non inizieranno a funzionare fino a quando il thread corrente non rilascerà il processore.



Ogni task ha sempre l’“impressione” di avere il processore a propria disposizione, ma il tempo di CPU viene sempre suddiviso tra tutti i task, tranne quando effettivamente il programma sta funzionando su un computer multiprocessore. Uno dei vantaggi principali del threading è che consente di astrarre da questo livello, di conseguenza il vostro codice non avrà necessità di sapere su quante CPU sta realmente funzionando. Quindi l'utilizzo dei thread è un metodo per creare programmi scalabili in modo trasparente: se un programma sta funzionando troppo lentamente, per accelerarlo basterà aggiungere processori al computer. In definitiva, le elaborazioni multitasking e multithreading sono i metodi più efficaci per utilizzare i sistemi multiprocessore.

### **Definizione dei task**

Considerato che un thread pilota un task, deve avere un modo per descrivere quel task, e questo gli viene fornito dall’interfaccia **Runnable**. Per definire un task basta implementare **Runnable** e scrivere un metodo **run()** per fare in modo che il task esegua quanto richiestogli.

Per esempio, il task **LiftOff** del codice seguente visualizza un conto alla rovescia.

```
//: concurrency/LiftOff.java
// Dimostrazione dell'interfaccia Runnable.

public class LiftOff implements Runnable {
    protected int countDown = 10; // Predefinito
    private static int taskCount = 0;
    private final int id = taskCount++;
    public LiftOff() {}
    public LiftOff(int countDown) {
        this.countDown = countDown;
    }
    public String status() {
        return "#" + id + "(" +
            (countDown > 0 ? countDown : "LiftOff!") + "), ";
    }
    public void run() {
        while(countDown-- > 0) {
            System.out.print(status());
        }
    }
}
```



```
    Thread.yield();
}
}
}
} //:/~
```

L'identificativo **id** distingue istanze multiple del task e viene definito come **final** poiché non è previsto che subisca modifiche dopo essere stato inizializzato.

Solitamente il metodo **run()** di un task ha qualche tipo di iterazione che continua finché il task non è più necessario, pertanto occorre impostare la condizione di uscita da questo ciclo: una possibilità è il semplice **return** da **run()**. Spesso il metodo **run()** è strutturato in forma di ciclo infinito; questo significa che, fatti salvi alcuni fattori che ne causano l'interruzione, **run()** continuerà per un tempo indefinito. Nel prosieguo del capitolo vedrete come concludere i task in modo sicuro.

La chiamata al metodo **static Thread.yield()** all'interno di **run()** è uno spunto per implementare il cosiddetto *thread scheduler* (letteralmente *pianificatore di thread*), vale a dire la parte del meccanismo di threading di Java che "commuta" la CPU da un thread all'altro. Questo metodo è facoltativo, ma viene utilizzato in questo esempio poiché produce un output più interessante, offrendovi maggiori probabilità di osservare la prova del passaggio da un thread all'altro.

Nel seguente esempio il metodo **run()** del task non è guidato da un thread separato, ma viene chiamato direttamente in **main()**; tenete presente che, in realtà, si sta *sempre* utilizzando un thread, quello che viene comunque assegnato a **main()**.

```
//: concurrency/MainThread.java

public class MainThread {
    public static void main(String[] args) {
        LiftOff launch = new LiftOff();
        launch.run();
    }
} /* Output:
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1),
#0(LiftOff!),
*//:/~
```



Quando create una classe per ereditarietà da **Runnable** dovete dotarla di un metodo **run()**, che tuttavia non ha nulla di speciale e non offre implicite funzionalità di threading. Per implementare il threading dovete espressamente collegare un task a un thread.

### **Classe Thread**

La tecnica tradizionale per trasformare un oggetto **Runnable** in un task funzionante consiste nel passarlo a un costruttore **Thread**. Il seguente esempio mostra come pilotare un oggetto **LiftOff** mediante un **Thread**.

```
//: concurrency/BasicThreads.java
// L'utilizzo piu' semplice della classe Thread.

public class BasicThreads {
    public static void main(String[] args) {
        Thread t = new Thread(new LiftOff());
        t.start();
        System.out.println("Waiting for LiftOff");
    }
} /* Output: (90% match)
Waiting for LiftOff
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1),
#0(LiftOff!),
*///:~
```

Un costruttore di **Thread** necessita di un oggetto **Runnable**. La chiamata al metodo **start()** di un oggetto **Thread()** esegue l'inizializzazione richiesta dal thread, poi chiama il metodo **run()** di **Runnable** per avviare il task nel nuovo thread. Benché **start()** sembri chiamare un metodo di lunga durata, come potete vedere dall'output il metodo **start()** ritorna rapidamente: infatti, il messaggio "Waiting for LiftOff" appare prima del termine del conto alla rovescia.

In realtà avete eseguito una chiamata al metodo **LiftOff.run()** e quel metodo non è ancora terminato, ma dal momento che **LiftOff.run()** viene eseguito da un diverso thread, potete svolgere altre operazioni nel thread di **main()**; tenete presente che questa capacità non si limita al **thread** di **main()**: qualsiasi thread può avviare un altro. A questo punto il programma sta eseguendo nello stesso tempo due metodi, **main()** e **LiftOff.run()**. Il metodo **run()** è il codice che viene eseguito "simultaneamente" con gli altri thread del programma.



Potete facilmente aggiungere più thread per pilotare più task. Nell'esempio seguente, osservate come tutti i task funzionino di concerto.<sup>5</sup>

```
//: concurrency/MoreBasicThreads.java
// Aggiunta di altri thread.

public class MoreBasicThreads {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new Thread(new LiftOff()).start();
        System.out.println("Waiting for LiftOff");
    }
} /* Output: (Esempio)
Waiting for LiftOff
#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8),
#3(8), #4(8), #0(7), #1(7), #2(7), #3(7), #4(7), #0(6),
#1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5), #3(5),
#4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3),
#2(3), #3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2),
#0(1), #1(1), #2(1), #3(1), #4(1), #0(LiftOff!),
#1(LiftOff!), #2(LiftOff!), #3(LiftOff!), #4(LiftOff!),
*///:-
```

L'output mostra che l'esecuzione dei vari task è "mescolata" a causa della permutazione dei thread, permutazione che viene gestita automaticamente dal *thread scheduler*. Se lavorate su un computer multiprocessore, il pianificatore si occuperà di distribuire i thread tra le varie CPU.

Il risultato di un'esecuzione di questo programma sarà differente da un altro, poiché il meccanismo di pianificazione dei thread non è deterministico; infatti, tra una versione di JDK e la successiva potreste notare differenze considerevoli nell'output di questo semplice programma. Per esempio, una versione di JDK potrebbe non eseguire il time slicing di frequente, pertanto il thread 1 potrebbe terminare per primo; a quel punto entrerebbero in azione le iterazioni del thread 2 ecc.

---

5. In questo caso, un solo thread (`main()`) sta creando tutti i thread `LiftOff`. Qualora abbiate diversi thread che creano quelli di `LiftOff`, invece, è possibile che più di un `LiftOff` abbia lo stesso `id`. Comprenderete i motivi di questo comportamento nel prosieguo del capitolo.



Questo equivale in pratica a chiamare una routine che esegue tutti i cicli immediatamente, salvo che attivare tutti quei thread sarebbe più oneroso. Le versioni di JDK più recenti sembrano avere una migliore funzionalità di time slicing, permettendo a ogni thread di operare con maggiore regolarità. In genere questo tipo di cambiamenti nel comportamento di JDK non era segnalato da Sun, quindi non è possibile ipotizzare che il comportamento di threading sia costante: di conseguenza, quando scrivete codice di thread l'approccio migliore è quello conservativo, teso a mantenere la massima prudenza.

Quando `main()` crea gli oggetti `Thread` non sta intercettando i riferimenti di nessuno di essi. Questo comportamento farebbe di tali oggetti i candidati ideali per la garbage collection, se essi non fossero `Thread`. Ogni `Thread` si “autoregistra”, in modo che il suo riferimento sia conservato, e il garbage collector non potrà prenderlo in carico fino a quando il task non sarà uscito dal proprio metodo `run()` e terminato. Potete vedere dall'output che i task si stanno avviando effettivamente a conclusione, pertanto un thread crea un thread di esecuzione separato che persiste dopo il termine della chiamata a `start()`.

**Esercizio 1** (2) Implementate una classe `Runnable`. All'interno del metodo `run()` visualizzate un messaggio, poi chiamate `yield()`; ripetete queste operazioni per tre volte, quindi ritornate da `run()`. Inserite un messaggio di avvio nel costruttore e un messaggio di fine quando l'operazione termina. Create un certo numero di questi task e pilotateli per mezzo dei thread.

**Esercizio 2** (2) Prendendo come esempio il codice di `generics/Fibonacci.java`, create un task che produce una sequenza di `n` numeri di Fibonacci, dove `n` è il valore fornito al costruttore del task. Create un certo numero di questi task e pilotateli per mezzo dei thread.

## **Utilizzo di Executor**

Gli `Executor` della libreria `java.util.concurrent` di Java SE5 semplificano la programmazione concorrente gestendo automaticamente gli oggetti `Thread`. Gli “esecutori” forniscono un “livello di indirezione” tra un client e l'esecuzione di un task: anziché essere un client a eseguire direttamente un task, questo compito è svolto da un oggetto intermedio. Gli `Executor` consentono di portare avanti l'esecuzione di task asincroni senza dover gestire il ciclo di vita dei thread, e sono l'approccio consigliato per avviare i task in Java SE5/6.<sup>6</sup>

---

6. In informatica, “indirezione” è la capacità di fare riferimento a un oggetto mediante un nome, un riferimento o un contenitore, anziché tramite il suo valore.



In `MoreBasicThreads.java` è possibile utilizzare `Executor` invece di creare esplicitamente degli oggetti `Thread`. Un oggetto `LiftOff` sa come eseguire un'operazione specifica; come il design pattern *Command*, espone un singolo metodo da eseguire. Un `ExecutorService`, vale a dire un `Executor` con un ciclo di vita di servizio, quale lo `shutdown`, sa come implementare il contesto adatto per eseguire gli oggetti `Runnable`. Nell'esempio seguente `CachedThreadPool` crea un thread per ogni thread; notate che un oggetto `ExecutorService` viene creato utilizzando un metodo `static` della classe `Executors`, che determina il tipo di `Executor`.

```
//: concurrency/CachedThreadPool.java
import java.util.concurrent.*;

public class CachedThreadPool {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new LiftOff());
        exec.shutdown();
    }
} /* Output: (Esempio)
#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8),
#2(8), #3(8), #4(8), #0(6), #1(7), #2(7), #3(7), #4(7),
#0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5), #2(5),
#3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2),
#1(3), #2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2),
#4(2), #0(LiftOff!), #1(1), #2(1), #3(1), #4(1),
#1(LiftOff!), #2(LiftOff!), #3(LiftOff!), #4(LiftOff!),
*///:-
```

Spesso, per creare e gestire tutte le operazioni nella vostra applicazione potete servirvi di un solo `Executor`.

La chiamata a `shutdown()` impedisce l'invio di nuove operazioni all'`Executor` corrente. Il thread corrente, in questo caso quello che pilota `main()`, continuerà a eseguire tutti i task sottoposti prima della chiamata a `shutdown()`. Il programma terminerà non appena saranno conclusi tutti i task in `Executor`.

Potete sostituire facilmente la classe `CachedThreadPool` utilizzata nell'esempio precedente con un tipo di `Executor` diverso. `FixedThreadPool`, per esem-



pio, sfrutta un insieme limitato di thread per eseguire i task che gli sono sottoposti.

```
//: concurrency/FixedThreadPool.java
import java.util.concurrent.*;

public class FixedThreadPool {
    public static void main(String[] args) {
        // L'argomento del costruttore e' il numero di thread:
        ExecutorService exec = Executors.newFixedThreadPool(5);
        for(int i = 0; i < 5; i++)
            exec.execute(new LiftOff());
        exec.shutdown();
    }
} /* Output: (Esempio)
#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8),
#2(8), #3(8), #4(8), #0(6), #1(7), #2(7), #3(7), #4(7),
#0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5), #2(5),
#3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2),
#1(3), #2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2),
#4(2), #0(LiftOff!), #1(1), #2(1), #3(1), #4(1),
#1(LiftOff!), #2(LiftOff!), #3(LiftOff!), #4(LiftOff!),
*///:~
```

Con **FixedThreadPool** l'onerosa allocazione del thread viene eseguita una sola volta, subito all'inizio, limitando così il numero di thread. Questo consente di risparmiare tempo poiché non dovete creare continuamente thread per ogni task; inoltre, in un sistema event-driven, i gestori degli eventi che richiedono i thread possono essere serviti più in fretta semplicemente prelevando i thread dall'insieme a disposizione, il cosiddetto *pool*. Non avrete modo di esaurire le risorse disponibili poiché **FixedThreadPool** utilizza un numero prestabilito di oggetti **Thread**.

Notate che in ogni pool, quando possibile, i thread esistenti vengono riutilizzati.

Anche se in questo libro utilizzerete i **CachedThreadPool**, dovreste valutare la possibilità di impiegare i **FixedThreadPool** nel codice di produzione. In genere un **CachedThreadPool** crea l'esatto numero di thread richiesti durante l'esecuzione di un programma, smettendo poi di crearne di nuovi per rici-



clare quelli esistenti. Per questo motivo, come prima scelta è opportuno che privilegiate l'**Executor**, e soltanto se questo meccanismo causasse problemi dovrà passare a un **FixedThreadPool**.

Un **SingleThreadExecutor** equivale a un **FixedThreadPool** con dimensione pari a un thread.<sup>7</sup>

**SingleThreadExecutor** è utile per qualsiasi oggetto che opera in modo continuativo in un altro thread, un task caratterizzato da una durata particolarmente lunga qual è, per esempio, quello destinato a rimanere in attesa di nuove connessioni su un socket. **SingleThreadExecutor** è anche utile per brevi task che volete vengano eseguiti in un thread, relativi per esempio agli aggiornamenti di un log locale o remoto, o per thread di tipo *event-dispatching*.

Se a un **SingleThreadExecutor** vengono sottoposti diversi task, questi saranno accodati e ciascuno verrà portato a completamento prima dell'avvio del task successivo, utilizzando lo stesso thread. Nell'esempio che segue ogni task viene completato nell'ordine in cui è stato sottoposto, prima che sia avviato quello seguente. Per questo motivo **SingleThreadExecutor** serializza i task che gli sono inviati e mantiene una propria coda nascosta di task in attesa.

```
//: concurrency/SingleThreadExecutor.java
import java.util.concurrent.*;

public class SingleThreadExecutor {
    public static void main(String[] args) {
        ExecutorService exec =
            Executors.newSingleThreadExecutor();
        for(int i = 0; i < 5; i++)
            exec.execute(new LiftOff());
        exec.shutdown();
    }
} /* Output:
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2),
#0(1), #0(LiftOff!), #1(9), #1(8), #1(7), #1(6), #1(5),
#1(4), #1(3), #1(2), #1(1), #1(LiftOff!), #2(9), #2(8),
#2(7), #2(6), #2(5), #2(4), #2(3), #2(2), #2(1),
```

7. **SingleThreadExecutor** offre anche un'importante garanzia di concorrenza che non è disponibile negli altri due pool: impedisce che due task siano eseguiti simultaneamente. Questa caratteristica modifica i requisiti di blocco dei task, un argomento che sarà trattato nel prossimo capitolo.



```
#2(LiftOff!), #3(9), #3(8), #3(7), #3(6), #3(5), #3(4),
#3(3), #3(2), #3(1), #3(LiftOff!), #4(9), #4(8), #4(7),
#4(6), #4(5), #4(4), #4(3), #4(2), #4(1), #4(LiftOff!),
*///:~
```

Come ulteriore esempio, supponete di avere un certo numero di thread per eseguire task che operino sul filesystem. Potrete eseguire questi task con **SingleThreadExecutor**, per accertarvi che solo un task per volta venga eseguito da ogni thread. In questo modo non dovrete occuparvi della sincronizzazione delle risorse condivise, e nello stesso tempo non farete scendere di colpo le prestazioni del filesystem. A volte, una soluzione migliore può essere la sincronizzazione della risorsa, un argomento che vedrete in dettaglio nel prosieguo del capitolo; in ogni caso, **SingleThreadExecutor** consente anche di evitare la difficoltà di definire una coordinazione perfetta anche per la creazione di un semplice prototipo. Serializzando i task eliminate la necessità di serializzare gli oggetti.

**Esercizio 3 (1)** Ripetete l'Esercizio 1 utilizzando i diversi tipi di esecutori presentati in questo paragrafo.

**Esercizio 4 (1)** Ripetete l'Esercizio 2 utilizzando i diversi tipi di esecutori presentati in questo paragrafo.

### Ottenere valori di ritorno dai task

L'interfaccia **Runnable** rappresenta un task separato, che esegue un'attività ma non restituisce alcun valore. Se desiderate che l'esecuzione di un task restituisca un valore, invece di **Runnable** potrete implementare **Callable**: introdotta in Java SE5, questa interfaccia è un generico con un parametro di tipo che rappresenta il valore restituito dal metodo **call()**, anziché dal metodo **run()** tipico dell'interfaccia **Runnable**. Il metodo **call()** deve essere invocato tramite il metodo **submit()** di **ExecutorService**, come in questo semplice esempio.

```
//: concurrency/CallableDemo.java
import java.util.concurrent.*;
import java.util.*;

class TaskWithResult implements Callable<String> {
    private int id;
    public TaskWithResult(int id) {
        this.id = id;
    }
}
```



```
public String call() {
    return "result of TaskWithResult " + id;
}
}

public class CallableDemo {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        ArrayList<Future<String>> results =
            new ArrayList<Future<String>>();
        for(int i = 0; i < 10; i++)
            results.add(exec.submit(new TaskWithResult(i)));
        for(Future<String> fs : results)
            try {
                // get() rimane in blocco fino al completamento:
                System.out.println(fs.get());
            } catch(InterruptedException e) {
                System.out.println(e);
                return;
            } catch(ExecutionException e) {
                System.out.println(e);
            } finally {
                exec.shutdown();
            }
    }
} /* Output:
Result of TaskWithResult 0
Result of TaskWithResult 1
Result of TaskWithResult 2
Result of TaskWithResult 3
Result of TaskWithResult 4
Result of TaskWithResult 5
Result of TaskWithResult 6
Result of TaskWithResult 7
Result of TaskWithResult 8
Result of TaskWithResult 9
*///:-
```



Il metodo **submit()** produce un oggetto **Future**, parametrizzato per il tipo di risultato restituito da **Callable**. Potete utilizzare il metodo **isDone()** di **Future** per verificare se l'operazione è stata completata. Al termine del task potete intercettare l'eventuale risultato chiamando il metodo **get()**; avete anche la possibilità di chiamare **get()** senza eseguire preventivamente la verifica con **isDone()**: in questo caso, tuttavia, **get()** rimarrà bloccato finché il risultato non sarà disponibile.

Potete anche chiamare **get()** con un valore di timeout, oppure **isDone()** per controllare se l'operazione è terminata, prima di tentare di chiamare **get()** per intercettare il risultato.

Il metodo sovraccarico **Executors.callable()** accetta oggetti **Runnable** e produce un **Callable**. **ExecutorService** possiede metodi “**invoke**”, che producono collezioni di oggetti **Callable**.

**Esercizio 5** (2) Modificate l'Esercizio 2 in modo che il task sia un oggetto **Callable** che calcola e visualizza la serie di Fibonacci. Create diversi task e visualizzate i risultati.

### *Messa in pausa con sleep()*

Una semplice tecnica utile per intervenire sul comportamento dei task consiste nel metodo **sleep()**, che sospende l'esecuzione di quel task per un determinato periodo. Se nella classe **LiftOff** sostituite la chiamata **yield()** con una chiamata **sleep()**, otterrete il risultato mostrato di seguito.

```
//: concurrency/SleepingTask.java
// Chiamata a sleep() per mettere in pausa il task.
import java.util.concurrent.*;

public class SleepingTask extends LiftOff {
    public void run() {
        try {
            while(countDown-- > 0) {
                System.out.print(status());
                // Vecchio stile:
                // Thread.sleep(100);
                // Stile Java SE5/6:
                TimeUnit.MILLISECONDS.sleep(100);
            }
        }
    }
}
```



```
        } catch(InterruptedException e) {
            System.err.println("Interrupted");
        }
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new SleepingTask());
        exec.shutdown();
    }
} /* Output:
#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8),
#3(8), #4(8), #0(7), #1(7), #2(7), #3(7), #4(7), #0(6),
#1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5), #3(5),
#4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3),
#2(3), #3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2),
#0(1), #1(1), #2(1), #3(1), #4(1), #0(LiftOff!),
#1(LiftOff!), #2(LiftOff!), #3(LiftOff!), #4(LiftOff!),
*///:-
```

La chiamata a `sleep()` può sollevare un'**InterruptedException** che viene intercettata in `run()`. Poiché le eccezioni non si propagheranno da un thread all'altro per ritornare a `main()`, dovete gestire a livello locale tutte le eccezioni che si presentassero all'interno di un task.

Java SE5 ha introdotto una versione di `sleep()` più esplicita, come componente della classe **TimeUnit**, utilizzata nell'esempio precedente. Questo metodo migliora la leggibilità del codice consentendo di specificare le unità di tempo di `sleep()`. Come vedrete nel prosieguo del capitolo, **TimeUnit** può anche essere utilizzata per eseguire conversioni.

A seconda della piattaforma potreste notare che l'esecuzione dei task è "perfettamente distribuita": da zero a quattro, poi di nuovo a zero. Questo ordinamento è perfettamente logico poiché dopo ciascuna istruzione di visualizzazione ogni task si mette in pausa, consentendo al pianificatore di passare a un altro thread per pilotare un altro task. Tuttavia questo comportamento sequenziale si basa sul meccanismo di threading di base, diverso per ogni sistema operativo, di conseguenza non potete farvi affidamento. Se dovete intervenire sull'ordine d'esecuzione dei task, l'approccio migliore risiede nell'utilizzo dei controlli di sincronizzazione, descritti in seguito; in alcuni casi è



persino preferibile non ricorrere ai thread, scrivendo procedure cooperative che controllino direttamente la successione dei task nell'ordine desiderato.

**Esercizio 6** (2) Create un task che rimanga in pausa per un periodo di tempo, determinato casualmente, compreso tra 1 e 10 secondi, quindi visualizzate il tempo di sleep e uscite dal task. Create ed eseguite un certo numero di questi task, equivalente a un parametro fornito sulla riga di comando.

### Priorità

La priorità di un thread ne trasferisce l'importanza al pianificatore. Sebbene l'ordine in cui la CPU esegue un insieme di thread sia indeterminato, il pianificatore cercherà di eseguire per primo il thread in attesa con il livello di priorità più elevato. Questo non significa, però, che i thread con priorità inferiore non verranno eseguiti, pertanto non potrà verificarsi una situazione di "stallo" dovuta alle priorità: la condizione di bassa priorità significa soltanto che i thread verranno eseguiti con minore frequenza.

Nella maggior parte dei casi l'esecuzione dei thread dovrebbe avvenire secondo la priorità predefinita, che è generalmente sconsigliabile modificare.

Questo è un esempio che dimostra il funzionamento dei livelli di priorità. Potete determinare la priorità corrente di un thread ricorrendo al metodo `getPriority()` e modificarla in qualunque momento con `setPriority()`.

```
//: concurrency/SimplePriorities.java
// Dimostra l'uso delle priorità dei thread.
import java.util.concurrent.*;

public class SimplePriorities implements Runnable {
    private int countDown = 5;
    private volatile double d; // Nessuna ottimizzazione
    private int priority;
    public SimplePriorities(int priority) {
        this.priority = priority;
    }
    public String toString() {
        return Thread.currentThread() + ": " + countDown;
    }
    public void run() {
```



```
Thread.currentThread().setPriority(priority);
while(true) {
    // Un'operazione onerosa che puo' essere interrotta:
    for(int i = 1; i < 100000; i++) {
        d += (Math.PI + Math.E) / (double)i;
        if(i % 1000 == 0)
            Thread.yield();
    }
    System.out.println(this);
    if(--countDown == 0) return;
}
}

public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
        exec.execute(
            new SimplePriorities(Thread.MIN_PRIORITY));
    exec.execute(
        new SimplePriorities(Thread.MAX_PRIORITY));
    exec.shutdown();
}
} /* Output: (70% match)
Thread[pool-1-thread-6,10,main]: 5
Thread[pool-1-thread-6,10,main]: 4
Thread[pool-1-thread-6,10,main]: 3
Thread[pool-1-thread-6,10,main]: 2
Thread[pool-1-thread-6,10,main]: 1
Thread[pool-1-thread-3,1,main]: 5
Thread[pool-1-thread-2,1,main]: 5
Thread[pool-1-thread-1,1,main]: 5
Thread[pool-1-thread-5,1,main]: 5
Thread[pool-1-thread-4,1,main]: 5
...
*///:~
```

Il codice sfrutta la variante sovrascritta `toString()`, ovvero il metodo `Thread.toString()`, per visualizzare il nome del thread, il livello di priorità e il "grup-



po” cui appartiene il thread corrente. Nel costruttore potete impostare il nome del thread, in questo caso generato automaticamente come **pool-1-thread-1**, **pool-1-thread-2** ecc. Il metodo sovrascritto **toString()** mostra anche il conto alla rovescia del task, che ne misura l'esecuzione. Tenete presente che è possibile ottenere un riferimento all'oggetto **Thread** che sta pilotando un task, chiamando **Thread.currentThread()** all'interno del task stesso.

Notate che la priorità dell'ultimo thread è al livello più elevato, mentre tutti gli altri thread sono al livello più basso. L'impostazione della priorità avviene all'inizio del metodo **run()**; se avvenisse nel costruttore non funzionerebbe, dal momento che in quel momento l'**Executor** non ha ancora iniziato il task.

All'interno di **run()** vengono eseguite 100.000 ripetizioni di un calcolo piuttosto complesso a virgola mobile, che include la somma e la divisione di un **double**. La variabile **d** è **volatile**, per evitare tentativi di ottimizzazione da parte del compilatore. Senza questo calcolo non vedreste il risultato dell'impostazione dei livelli di priorità; come verifica, provate a commentare il ciclo **for** contenente i calcoli **double**. Con il calcolo, invece, al thread con **MAX\_PRIORITY** il pianificatore assegna una priorità maggiore: quantomeno, questo è il comportamento osservato su un sistema Windows XP. Malgrado la visualizzazione a console sia anch'essa un'operazione “onerosa” non permetterà di osservare i livelli di priorità, poiché la visualizzazione, a differenza di un calcolo matematico, non viene interrotta: in questo caso, infatti, le informazioni sullo schermo non sarebbero comprensibili. Il calcolo richiede un tempo sufficiente da permettere al pianificatore di entrare in azione, sposare i task e gestire le priorità, in modo da privilegiare i thread con livelli di priorità superiori. Tuttavia, per assicurare che avvenga un cambio di contesto vengono eseguite regolarmente chiamate a **yield()**.

Benché il JDK offra 10 livelli di priorità, non tutti trovano riscontro nei diversi sistemi operativi. Per esempio, Windows ha 7 livelli di priorità che non sono fissi, pertanto la corrispondenza sarà indeterminata; Solaris di Sun ha un numero di livelli pari a 231. L'unica tecnica che garantisca una certa portabilità consiste nell'impostare i livelli di priorità mediante le parole chiave **MAX\_PRIORITY**, **NORM\_PRIORITY** e **MIN\_PRIORITY**.

## ***Yielding***

Se avete la certezza che l'operazione di cui avete bisogno è già avvenuta regolarmente durante un'iterazione nel vostro metodo **run()**, potete segnalare al pianificatore che non è più necessario continuare, mettendo così la CPU a disposizione di un altro task. Questo “invito” a modificare le precedenze delle operazioni prende il nome di metodo **yield()** (letteralmente, diritto di precedenza): si tratta effettivamente soltanto di un invito, che non vi offre alcuna garanzia di

essere implementato. In pratica, quando chiamate il metodo **yield()** vi limitate a "raccomandare" che altri thread con la stessa priorità vengano eseguiti.

Il codice di **LiftOff.java**, che avete visto nel paragrafo "Definizione dei task", utilizza **yield()** per distribuire uniformemente l'attività di elaborazione tra i vari task di **LiftOff**: provate a commentare la chiamata **Thread.yield()** in **LiftOff.run()** e noterete la differenza. In genere, però, non è possibile contare sul metodo **yield()** per implementare seri controlli o messe a punto delle applicazioni; da questo punto di vista, spesso **yield()** è utilizzato in modo errato.

### **Thread demoni**

Un thread "demon" (*daemon*) è utilizzato per fornire a livello generale un servizio essenziale che, pur non essendo parte integrante del programma, rimane attivo finché il programma è in esecuzione. Pertanto, quando tutti i thread non daemon sono stati completati, il programma termina chiudendo tutti i demoni avviati dal processo; al contrario, se sono ancora in esecuzione thread non daemon il programma continua. È un thread non daemon, per esempio, quello che fa funzionare **main()**.

```
//: concurrency/SimpleDaemons.java
// I thread daemon non impediscono la chiusura del programma.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

public class SimpleDaemons implements Runnable {
    public void run() {
        try {
            while(true) {
                TimeUnit.MILLISECONDS.sleep(100);
                print(Thread.currentThread() + " " + this);
            }
        } catch(InterruptedException e) {
            print("sleep() interrupted");
        }
    }
    public static void main(String[] args) throws Exception {
        for(int i = 0; i < 10; i++) {
            Thread daemon = new Thread(new SimpleDaemons());
            daemon.setDaemon(true); // Da chiamare prima di start()
        }
    }
}
```



```
        daemon.start();
    }
    print("All daemons started");
    TimeUnit.MILLISECONDS.sleep(175);
}
} /* Output: (Esempio)
All daemons started
Thread[Thread-0,5,main] SimpleDaemons@530daa
Thread[Thread-1,5,main] SimpleDaemons@a62fc3
Thread[Thread-2,5,main] SimpleDaemons@89ae9e
Thread[Thread-3,5,main] SimpleDaemons@1270b73
Thread[Thread-4,5,main] SimpleDaemons@60aeb0
Thread[Thread-5,5,main] SimpleDaemons@16caf43
Thread[Thread-6,5,main] SimpleDaemons@66848c
Thread[Thread-7,5,main] SimpleDaemons@8813f2
Thread[Thread-8,5,main] SimpleDaemons@1d58aae
Thread[Thread-9,5,main] SimpleDaemons@83cc67
...
*///:-
```

Per impostare un demone dovete chiamare il metodo **setDaemon()** prima dell'avvio del thread.

Dal momento che sono attivi soltanto thread demoni, il programma termina senza impedimenti quando **main()** ha concluso la sua attività. Per consentirvi di osservare i risultati dell'avvio di tutti i thread daemon, il thread **main()** viene brevemente messo in pausa. Senza questo accorgimento vedreste soltanto alcuni risultati dell'avvio dei demoni; per verificare questo comportamento provate a chiamare **sleep()** con parametri di durata differenti.

**SimpleDaemons.java** crea oggetti **Thread** esplicativi per impostarne il flag "daemon". Potete personalizzare gli attributi (daemon, priorità, nome) dei thread creati dagli **Executor**, scrivendo una **ThreadFactory** dedicata.

```
//: net/mindview/util/DaemonThreadFactory.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
```



```
    Thread t = new Thread(r);
    t.setDaemon(true);
    return t;
}
} /* (Da eseguire per visualizzare l'output) */:-
```

L'unica differenza rispetto a una normale **ThreadFactory** è che questa variante imposta lo stato di daemon a **true**. A questo punto potete passare una nuova **DaemonThreadFactory** come argomento di **Executors.newCachedThreadPool()**.

```
//: concurrency/DaemonFromFactory.java
// Utilizzo di una Thread Factory per creare demoni.
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DaemonFromFactory implements Runnable {
    public void run() {
        try {
            while(true) {
                TimeUnit.MILLISECONDS.sleep(100);
                print(Thread.currentThread() + " " + this);
            }
        } catch(InterruptedException e) {
            print("Interrupted");
        }
    }
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool(
            new DaemonThreadFactory());
        for(int i = 0; i < 10; i++)
            exec.execute(new DaemonFromFactory());
        print("All daemons started");
        TimeUnit.MILLISECONDS.sleep(500); // Disattiva per il
                                         // tempo indicato
    }
} /* (Da eseguire per visualizzare l'output) */:-
```



Ognuno dei metodi di creazione **static ExecutorService** è sovraccarico per accettare un oggetto **ThreadFactory** che utilizzerà per generare i nuovi thread.

Migliorerete ulteriormente il programma realizzando un'utility **Daemon ThreadPoolExecutor**.

```
//: net/mindview/util/DaemonThreadPoolExecutor.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadPoolExecutor
extends ThreadPoolExecutor {
    public DaemonThreadPoolExecutor() {
        super(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS,
              new SynchronousQueue<Runnable>(),
              new DaemonThreadFactory());
    }
} //:-
```

Per ottenere i valori necessari per la chiamata al costruttore della classe di base, è sufficiente che esaminiate il codice sorgente di **Executors.java**.

Il metodo **isDaemon()** consente di scoprire se un thread è un daemon; in caso affermativo, tutti i thread che verranno prodotti da quel thread saranno automaticamente demoni, come dimostra il seguente esempio.

```
//: concurrency/Daemons.java
// I thread daemon danno origine ad altri demoni.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Daemon implements Runnable {
    private Thread[] t = new Thread[10];
    public void run() {
        for(int i = 0; i < t.length; i++) {
            t[i] = new Thread(new DaemonSpawn());
            t[i].start();
            printnb("DaemonSpawn " + i + " started, ");
        }
    }
}
```



```
for(int i = 0; i < t.length; i++)
    println("t[" + i + "].isDaemon() = " +
           t[i].isDaemon() + ", ");
while(true)
    Thread.yield();
}
}

class DaemonSpawn implements Runnable {
    public void run() {
        while(true)
            Thread.yield();
    }
}

public class Daemons {
    public static void main(String[] args) throws Exception {
        Thread d = new Thread(new Daemon());
        d.setDaemon(true);
        d.start();
        println("d.isDaemon() = " + d.isDaemon() + ", ");
        // Permette ai thread daemon di concludere
        // i rispettivi processi di avvio:
        TimeUnit.SECONDS.sleep(1);
    }
} /* Output: (Esempio)
d.isDaemon() = true, DaemonSpawn 0 started, DaemonSpawn 1
started, DaemonSpawn 2 started, DaemonSpawn 3 started,
DaemonSpawn 4 started, DaemonSpawn 5 started, DaemonSpawn 6
started, DaemonSpawn 7 started, DaemonSpawn 8 started,
DaemonSpawn 9 started, t[0].isDaemon() = true,
t[1].isDaemon() = true, t[2].isDaemon() = true,
t[3].isDaemon() = true, t[4].isDaemon() = true,
t[5].isDaemon() = true, t[6].isDaemon() = true,
t[7].isDaemon() = true, t[8].isDaemon() = true,
t[9].isDaemon() = true,
*///:~
```



In **main()**, il thread **Daemon** viene dapprima impostato alla modalità demon; poi viene creato un certo numero di thread, che non sono esplicitamente impostati a daemon, per dimostrare che si tratta comunque di demoni; a quel punto **Daemon** entra in un ciclo infinito che chiama **yield()** per assegnare il controllo ad altri processi. Tenete presente che i thread demoni termineranno i loro metodi **run()** senza eseguire le istruzioni **finally**.

```
//: concurrency/DaemonsDontRunFinally.java
// I thread daemon non eseguono la clausola finally
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class ADAemon implements Runnable {
    public void run() {
        try {
            print("Starting ADAemon");
            TimeUnit.SECONDS.sleep(1);
        } catch(InterruptedException e) {
            print("Exiting via InterruptedException");
        } finally {
            print("This should always run?");
        }
    }
}

public class DaemonsDontRunFinally {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new ADAemon());
        t.setDaemon(true);
        t.start();
    }
} /* Output:
Starting ADAemon
*///:~
```

Eseguendo il programma noterete che la clausola **finally** non viene eseguita, ma se commentate la chiamata a **setDaemon()** l'istruzione **finally** sarà correttamente eseguita.



Questo comportamento è corretto anche se probabilmente non corrisponde alle vostre aspettative, tenuto conto delle informazioni che vi sono state fornite a proposito della clausola **finally**. I demoni vengono interrotti “bruscamente” quando termina l’ultimo task non daemon: pertanto, non appena **main()** termina, la JVM interrompe di colpo tutti i daemon, senza alcuna formalità prevedibile.

Poiché non è possibile chiudere i daemon se non in modo brusco, raramente è una buona idea servirsi di questa tecnica. Di solito gli **Executor** non daemon sono una soluzione migliore, poiché consentono l’interruzione simultanea di tutti i task gestiti da un **Executor**: come vedrete nel prosieguo del capitolo, in questo caso l’interruzione avviene in modo ordinato.

**Esercizio 7** (2) Sperimentate con **Daemons.java** impostando differenti tempi di sleep, e prendete nota di quanto accade.

**Esercizio 8** (1) Modificate **MoreBasicThreads.java** in modo che tutti i thread siano demoni, e verificate che il programma si concluda non appena **main()** è in condizioni di terminare.

**Esercizio 9** (3) Modificate **SimplePriorities.java** in modo che una **ThreadFactory** personalizzata imposta le priorità dei thread.

### *Variazioni sul codice*

Negli esempi visti finora tutte le classi di task implementano **Runnable**. In casi molto semplici potete adottare l’approccio alternativo che prevede di ereditare direttamente da **Thread**, come nell’esempio seguente.

```
//: concurrency/SimpleThread.java
// Come ereditare direttamente dalla classe Thread.

public class SimpleThread extends Thread {
    private int countDown = 5;
    private static int threadCount = 0;
    public SimpleThread() {
        // Memorizza il nome del thread:
        super(Integer.toString(++threadCount));
        start();
    }
    public String toString() {
        return "#" + getName() + "(" + countDown + ")", ";
    }
}
```



```
public void run() {
    while(true) {
        System.out.print(this);
        if(--countDown == 0)
            return;
    }
}

public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
        new SimpleThread();
}

/* Output:
#1(5), #1(4), #1(3), #1(2), #1(1), #2(5), #2(4), #2(3), #2(2),
#2(1), #3(5), #3(4), #3(3), #3(2), #3(1), #4(5), #4(4), #4(3),
#4(2), #4(1), #5(5), #5(4), #5(3), #5(2), #5(1),
*///:~
```

Un costruttore **Thread** appropriato assegna nomi specifici agli oggetti **Thread**; questi nomi vengono poi recuperati in **toString()** per mezzo del metodo **getName()**.

Un'altra forma idiomatica che potreste trovare è quella **Runnable** autogestita.

```
//: concurrency/SelfManaged.java
// Un Runnable che contiene il proprio Thread di gestione.
```

```
public class SelfManaged implements Runnable {
    private int countDown = 5;
    private Thread t = new Thread(this);
    public SelfManaged() { t.start(); }
    public String toString() {
        return Thread.currentThread().getName() +
            "(" + countDown + ")", ";
    }
    public void run() {
        while(true) {
            System.out.print(this);
            if(--countDown == 0)
```



```
        return;
    }
}

public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
        new SelfManaged();
}

/* Output:
Thread-0(5), Thread-0(4), Thread-0(3), Thread-0(2), Thread-0(1),
Thread-1(5), Thread-1(4), Thread-1(3), Thread-1(2), Thread-1(1),
Thread-2(5), Thread-2(4), Thread-2(3), Thread-2(2), Thread-2(1),
Thread-3(5), Thread-3(4), Thread-3(3), Thread-3(2), Thread-3(1),
Thread-4(5), Thread-4(4), Thread-4(3), Thread-4(2), Thread-4(1),
*///:~
```

Questo approccio non è molto diverso da quello che eredita da **Thread**, salvo che la sintassi è meno lineare. Tuttavia, il fatto di implementare un'interfaccia consente di ereditare da una classe diversa, risultato che non è possibile ottenere ereditando da **Thread**.

Notate che il metodo **start()** è chiamato dal costruttore. Questo esempio è piuttosto semplice, pertanto probabilmente sicuro; tenete presente, tuttavia, che avviare i thread all'interno di un costruttore può essere molto difficile, poiché un altro task potrebbe attivarsi prima che il costruttore abbia completato le sue operazioni, permettendo così al thread di accedere all'oggetto in una condizione instabile. Questo è un altro motivo che giustifica la necessità di privilegiare gli **Executor** rispetto alla creazione di oggetti **Thread** esplicativi.

A volte è opportuno nascondere il codice di thread all'interno di una classe ricorrendo a una classe interna, come mostrato di seguito.

```
//: concurrency/ThreadVariations.java
// Creazione dei thread mediante le classi interne.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

// Utilizzo di una classe interna:
class InnerThread1 {
    private int countDown = 5;
    private Inner inner;
    private class Inner extends Thread {
```



```
Inner(String name) {
    super(name);
    start();
}
public void run() {
    try {
        while(true) {
            print(this);
            if(--countDown == 0) return;
            sleep(10);
        }
    } catch(InterruptedException e) {
        print("interrupted");
    }
}
public String toString() {
    return getName() + ":" + countDown;
}
}
public InnerThread1(String name) {
    inner = new Inner(name);
}
}

// Utilizzo di una classe interna anonima:
class InnerThread2 {
    private int countDown = 5;
    private Thread t;
    public InnerThread2(String name) {
        t = new Thread(name) {
            public void run() {
                try {
                    while(true) {
                        print(this);
                        if(--countDown == 0) return;
                        sleep(10);
                    }
                }
            }
        };
    }
}
```



```
        } catch(InterruptedException e) {
            print("sleep() interrupted");
        }
    }
    public String toString() {
        return getName() + ":" + countDown;
    }
};

t.start();
}

// Utilizzo di un'implementazione Runnable:
class InnerRunnable1 {
    private int countDown = 5;
    private Inner inner;
    private class Inner implements Runnable {
        Thread t;
        Inner(String name) {
            t = new Thread(this, name);
            t.start();
        }
        public void run() {
            try {
                while(true) {
                    print(this);
                    if(--countDown == 0) return;
                    TimeUnit.MILLISECONDS.sleep(10);
                }
            } catch(InterruptedException e) {
                print("sleep() interrupted");
            }
        }
        public String toString() {
            return t.getName() + ":" + countDown;
        }
    }
}
```



```
public InnerRunnable1(String name) {
    inner = new Inner(name);
}
}

// Utilizzo di un'implementazione Runnable anonima:
class InnerRunnable2 {
    private int countDown = 5;
    private Thread t;
    public InnerRunnable2(String name) {
        t = new Thread(new Runnable() {
            public void run() {
                try {
                    while(true) {
                        print(this);
                        if(--countDown == 0) return;
                        TimeUnit.MILLISECONDS.sleep(10);
                    }
                } catch(InterruptedException e) {
                    print("sleep() interrupted");
                }
            }
        });
        public String toString() {
            return Thread.currentThread().getName() +
                ":" + countDown;
        }
        name;
    }
    t.start();
}

// Un metodo separato che esegue codice sotto forma di task:
class ThreadMethod {
    private int countDown = 5;
    private Thread t;
    private String name;
    public ThreadMethod(String name) { this.name = name; }
```



```
public void runTask() {
    if(t == null) {
        t = new Thread(name) {
            public void run() {
                try {
                    while(true) {
                        print(this);
                        if(--countDown == 0) return;
                        sleep(10);
                    }
                } catch(InterruptedException e) {
                    print("sleep() interrupted");
                }
            }
            public String toString() {
                return getName() + ":" + countDown;
            }
        };
        t.start();
    }
}

public class ThreadVariations {
    public static void main(String[] args) {
        new InnerThread1("InnerThread1");
        new InnerThread2("InnerThread2");
        new InnerRunnable1("InnerRunnable1");
        new InnerRunnable2("InnerRunnable2");
        new ThreadMethod("ThreadMethod").runTask();
    }
} /* (Da eseguire per visualizzare l'output) */:-~
```

**InnerThread1** crea una classe interna che estende **Thread**, e crea un'istanza di questa classe all'interno del costruttore. Questa tecnica ha senso qualora la classe interna disponga di funzionalità speciali (nuovi metodi) che vorreste poter richiamare da altri. In genere, però, l'unico motivo per generare un thread è la possibilità di sfruttare le funzionalità di **Thread**, pertanto non vi



occorrerà creare una classe interna standard. **InnerThread2** suggerisce un'alternativa: una sottoclasse interna anonima di **Thread** creata all'interno del costruttore e sottoposta a upcast a un riferimento **T** di **Thread**. Se altri metodi della classe dovessero accedere a **t** potranno farlo tramite l'interfaccia **Thread**, senza conoscere il tipo esatto dell'oggetto.

La terza e la quarta classe nell'esempio riprendono le prime due classi, ma impiegano l'interfaccia **Runnable** anziché la classe **Thread**.

La classe **ThreadMethod** mostra la creazione di un thread all'interno di un metodo. Chiamate il metodo quando siete pronti a eseguire il thread e il metodo ritornerà dopo aver avviato il thread: se il thread deve soltanto eseguire un'operazione ausiliaria (non fondamentale) della classe, questa è probabilmente la tecnica più adatta per avviare un thread all'interno del costruttore della classe.

**Esercizio 10 (4)** Modificate l'Esercizio 5 seguendo l'esempio della classe **ThreadMethod**, in modo che **runTask()** accetti un argomento corrispondente alla quantità di numeri di Fibonacci da produrre; ogni volta che viene chiamato il metodo **runTask()** dovrà essere restituito l'oggetto **Future** prodotto dalla chiamata a **submit()**.

## Terminologia

Come avete visto nel paragrafo precedente, Java consente di scegliere come implementare i programmi concorrenti, e tali scelte possono generare confusione. Spesso il problema è legato alla terminologia impiegata per descrivere la tecnologia di programmazione concorrente, soprattutto nei casi che coinvolgono i thread.

Ormai sapete che esiste una differenza tra il task in corso di esecuzione e il thread che lo pilota; questa distinzione è evidente soprattutto nelle librerie Java, poiché effettivamente non avete alcun controllo sulla classe **Thread**, ed è ancor più netta per gli "esecutori", che automaticamente si prendono cura della creazione e della gestione dei thread. Voi create il task e in qualche modo gli collegate un thread, che avrà il compito di "pilotarlo".

In Java la classe di **Thread** in sé non esegue alcunché, limitandosi a guidare il task assegnatole. La letteratura sul threading, però, si ostina a utilizzare frasi quali "il thread esegue una certa azione". L'impressione che ne deriva è che il thread sia il task: questa impressione può essere così forte da far ritenere che esista un'implicita relazione di tipo "è-un" (is-a), e quindi dare per scontato che un task possa ereditare da **Thread**. A questo aggiungete la pessima scelta di nomenclatura per l'interfaccia **Runnable**, che molti ritengono sarebbe stato meglio chiamare "**Task**". Se l'interfaccia è chiaramente un semplice contenitore generico che include metodi, il nome "questo-è-ciò-che-fa-able"



andrà benissimo, ma se intende esprimere un concetto più elevato, come **Task**, allora sarà più utile un nome che lo richiami esplicitamente.

Il problema è che i livelli di astrazione sono miscelati. Dal punto di vista concettuale si vuole creare un task che operi indipendentemente dagli altri, pertanto si vorrebbe definirlo, ordinargli di funzionare e non preoccuparsi dei particolari. Dal punto di vista materiale, però, la generazione dei thread può essere onerosa, di conseguenza è opportuno, una volta creati, mantenerli e gestirli. Dal punto di vista dell'implementazione ha quindi significato separare i task dai thread. Inoltre il threading di Java è basato sull'approccio a basso livello dei *pthread*, derivato da C: un approccio nel quale si è completamente immersi e del quale occorre capire a fondo ogni singolo dettaglio. Una parte di questa natura a basso livello è passata nell'implementazione Java, pertanto per mantenersi a un livello di astrazione più alto dovete rispettare il massimo rigore nella scrittura del codice. Nel corso di questo capitolo l'autore proverà a dimostrarvi questa disciplina.

Al fine di evitare situazioni confuse, nei prossimi argomenti si cercherà, per quanto possibile, di attenersi al termine "task" per descrivere l'operazione da effettuare, riservando il termine "thread" al solo meccanismo che pilota il task stesso. Quindi, se state discutendo di un sistema a livello concettuale potrete semplicemente utilizzare il termine "task" senza accennare al "meccanismo di guida".

### **Collegamento dei thread**

Un thread può chiamare il metodo **join()** su un altro thread per aspettare il completamento del secondo thread prima di continuare: in pratica, se un thread chiama **t.join()** su un altro thread **t**, il chiamante rimane sospeso fino al completamento del thread **t** obiettivo, vale a dire fino a quando **t.isAlive()** non diventa **false**.

Potete anche chiamare **join()** con un argomento di timeout, espresso in milisecondi o nanosecondi, per fare in modo che se il thread obiettivo non termina entro un certo tempo, la chiamata **join()** ritorni in qualsiasi caso.

La chiamata a **join()** può essere abbandonata chiamando **interrupt()** sul thread chiamante, per cui dovete prevedere una clausola **try-catch**.

Tutte queste operazioni sono illustrate nell'esempio seguente.

```
//: concurrency/Joining.java
// Comprendere join().
import static net.mindview.util.Print.*;
```



```
class Sleeper extends Thread {  
    private int duration;  
    public Sleeper(String name, int sleepTime) {  
        super(name);  
        duration = sleepTime;  
        start();  
    }  
    public void run() {  
        try {  
            sleep(duration);  
        } catch(InterruptedException e) {  
            print(getName() + " was interrupted. " +  
                  "isInterrupted(): " + isInterrupted());  
            return;  
        }  
        print(getName() + " has awakened.");  
    }  
}  
  
class Joiner extends Thread {  
    private Sleeper sleeper;  
    public Joiner(String name, Sleeper sleeper) {  
        super(name);  
        this.sleeper = sleeper;  
        start();  
    }  
    public void run() {  
        try {  
            sleeper.join();  
        } catch(InterruptedException e) {  
            print("Interrupted");  
        }  
        print(getName() + " join completed.");  
    }  
}
```



```
public class Joining {
    public static void main(String[] args) {
        Sleeper
            sleepy = new Sleeper("Sleepy", 1500),
            grumpy = new Sleeper("Grumpy", 1500);
        Joiner
            dopey = new Joiner("Dopey", sleepy),
            doc = new Joiner("Doc", grumpy);
        grumpy.interrupt();
    }
} /* Output:
Grumpy was interrupted. isInterrupted(): false
Doc join completed.
Sleepy has awakened.
Dopey join completed.
*///:~
```

Uno **Sleeper** è un thread che “dorme” (*sleep*) per un periodo di tempo specificato nel suo costruttore. In **run()** la chiamata a **sleep()** può terminare allo scadere del tempo, ma può anche essere interrotta. All’interno della clausola **catch** l’interruzione è segnalata, insieme al valore di **isInterrupted()**. Quando un altro thread chiama **interrupt()** sul thread corrente, viene impostato un flag per indicare che il thread è stato interrotto. Tuttavia tale flag viene ripristinato (annullato) non appena è intercettata l’eccezione, in modo che all’interno della clausola **catch** il risultato sia sempre **false**. Il flag è applicato in altre situazioni in cui un thread potrebbe esaminare, oltre a quello dell’eccezione, anche il proprio stato di interruzione.

Un **Joiner** è un task che rimane in attesa del “risveglio” (riattivazione) di uno **Sleeper**, che attiva chiamando **join()** sull’oggetto **Sleeper**. In **main()**, a ogni **Sleeper** corrisponde un **Joiner** e, come potete vedere dall’output, se lo **Sleeper** viene interrotto o si conclude normalmente, il **Joiner** termina insieme al suo **Sleeper**. Tenete presente che la libreria **java.util.concurrent** di Java SE5 contiene altri oggetti quali **CyclicBarrier**, presentato nel prosieguo del capitolo, che potrebbero essere più adatti di **join()**, incluso nella vecchia libreria di threading.

### ***Creazione di interfacce utente reattive***

Come si è detto, una delle ragioni che giustificano l’utilizzo del threading è la creazione di un’interfaccia di utente reattiva. Anche se le interfacce grafiche vere e proprie sono l’argomento del prossimo capitolo, l’esempio



seguinte è una semplice simulazione di un'interfaccia basata su console. L'esempio ha due varianti: la prima (**UnresponsiveUI**) rimane bloccata su un calcolo infinito e non arriverà mai a leggere l'input da console, mentre la seconda (**ResponsiveUI**) include lo stesso calcolo in un task, in modo da eseguire il calcolo e simultaneamente rimanere in attesa dell'input da console.

```
//: concurrency/ResponsiveUI.java
// Reattività dell'interfaccia utente.
// {RunByHand}

class UnresponsiveUI {
    private volatile double d = 1;
    public UnresponsiveUI() throws Exception {
        while(d > 0)
            d = d + (Math.PI + Math.E) / d;
        System.in.read(); // Non vi arriverà mai
    }
}

public class ResponsiveUI extends Thread {
    private static volatile double d = 1;
    public ResponsiveUI() {
        setDaemon(true);
        start();
    }
    public void run() {
        while(true) {
            d = d + (Math.PI + Math.E) / d;
        }
    }
    public static void main(String[] args) throws Exception {
        //! new UnresponsiveUI(); // Bisogna terminare questo
                               // processo con kill
        new ResponsiveUI();
        System.in.read();
```



```
System.out.println(d); // Visualizza il progresso  
// dell'operazione  
}  
} //:~
```

**UnresponsiveUI** esegue un calcolo all'interno di un ciclo **while** infinito, pertanto ovviamente non accederà mai alla riga di input della console: il compilatore è ingannato dal **while** condizionale e crede invece che la riga di input sia raggiungibile. Se togliete il commento alla riga che crea **UnresponsiveUI** dovrete terminare il processo per uscire.

Se volete realizzare un programma reattivo, inserite il calcolo all'interno di un metodo **run()** per renderlo *preempted*, ovvero "prioritario": quando premerete il tasto Invio, vedrete che il programma starà effettivamente eseguendo il calcolo mentre è in attesa del vostro input.

### ***Gruppi di thread***

Un gruppo di thread contiene una collezione di thread. Il significato dei gruppi di thread può riassumersi in un commento di Joshua Bloch (*Effective Java Programming Language Guide*, Addison-Wesley, 2001, pag. 211), l'ex progettista software di Sun che tra le altre realizzazioni ha contribuito a migliorare notevolmente la libreria di collezioni del JDK 1.2.

*"I gruppi di thread possono essere visti meglio in un'ottica di esperimento insoddisfacente, e potete semplicemente ignorare la loro esistenza."*

Se avete tempo ed energie da investire per cercare di comprendere l'effettiva validità dei gruppi di thread, come ha fatto l'autore, potreste chiedervi perché non vi sia alcuna comunicazione ufficiale da parte di Sun sull'argomento, e la stessa domanda potrebbe essere posta anche a proposito di numerosi cambiamenti che hanno interessato Java nel corso degli anni. L'economista e premio Nobel Joseph Stiglitz ha una filosofia di vita, chiamata *Theory of Escalating Commitment* (teoria scalare delle responsabilità), che potrebbe applicarsi perfettamente a questa e a molte altre situazioni che riguardano non soltanto Java, ma anche molti progetti ai quali l'autore, per esempio, ha partecipato come consulente.

*"Il costo di continui errori è sostentato dagli altri, mentre il costo legato all'ammissione degli errori è supportato soltanto da voi."*



## Intercettazione delle eccezioni

A causa della natura dei thread, non è possibile intercettare un'eccezione sollevata da un thread: una volta che l'eccezione sia stata prodotta dal metodo `run()` di un task, si propagherà fino alla console a meno che non prendiate gli opportuni accorgimenti per intercettare queste eccezioni "errabonde".

In passato l'intercettazione di tali eccezioni richiedeva l'utilizzo dei gruppi di thread, tuttavia da Java SE5 il problema può essere risolto con gli **Executor**; non avete quindi più bisogno di conoscere il meccanismo dei gruppi di thread, ma soltanto la capacità di leggere il vecchio codice Java. Per maggiori dettagli sulle funzionalità dei gruppi di thread potete consultare *Thinking in Java, seconda edizione*, scaricabile dal sito [www.mindview.net](http://www.mindview.net).

Osservate il codice del task seguente: solleva sempre un'eccezione che si propaga all'esterno del relativo metodo `run()`, con un `main()` che visualizza quanto accade in fase di esecuzione.

```
//: concurrency/ExceptionThread.java
// {ThrowsException}
import java.util.concurrent.*;

public class ExceptionThread implements Runnable {
    public void run() {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new ExceptionThread());
    }
} //:~
```

Dopo avere eliminato alcuni qualificatori, l'output sarà simile al seguente.

```
java.lang.RuntimeException
    at ExceptionThread.run(ExceptionThread.java:7)
    at ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:650)
    at ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:675)
    at java.lang.Thread.run(Thread.java:613)
```



L'inclusione del corpo del main in un blocco **try-catch** non offre alcun miglioramento.

```
//: concurrency/NaiveExceptionHandling.java
// {ThrowsException}
import java.util.concurrent.*;

public class NaiveExceptionHandling {
    public static void main(String[] args) {
        try {
            ExecutorService exec =
                Executors.newCachedThreadPool();
            exec.execute(new ExceptionThread());
        } catch(RuntimeException ue) {
            // Questa istruzione NON verrà' eseguita!
            System.out.println("Exception has been handled!");
        }
    }
} ///:~
```

Questo codice, infatti, fornisce lo stesso risultato dell'esempio precedente: un'eccezione non intercettata.

Per risolvere il problema occorre cambiare il modo in cui **Executor** produce i thread. La nuova interfaccia **Thread.UncaughtExceptionHandler** consente di collegare un gestore di eccezioni a ogni oggetto **Thread**. Il metodo **Thread.UncaughtExceptionHandler.uncaughtException()** viene chiamato automaticamente quando il thread sta terminando a seguito di un'eccezione non intercettata. Per utilizzare questo metodo, create un nuovo tipo di **ThreadFactory** che abbina un nuovo **Thread.UncaughtExceptionHandler** a ogni nuovo oggetto **Thread** creato da **ThreadFactory**, poi passate questa factory al metodo di **Executors** che crea un nuovo **ExecutorService**.

```
//: concurrency/CaptureUncaughtException.java
import java.util.concurrent.*;

class ExceptionThread2 implements Runnable {
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("run() by " + t);
```



```
        System.out.println(
            "eh = " + t.getUncaughtExceptionHandler());
        throw new RuntimeException();
    }
}

class MyUncaughtExceptionHandler implements
Thread.UncaughtExceptionHandler {
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println("caught " + e);
    }
}

class HandlerThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
        System.out.println(this + " creating new Thread");
        Thread t = new Thread(r);
        System.out.println("created " + t);
        t.setUncaughtExceptionHandler(
            new MyUncaughtExceptionHandler());
        System.out.println(
            "eh = " + t.getUncaughtExceptionHandler());
        return t;
    }
}

public class CaptureUncaughtException {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool(
            new HandlerThreadFactory());
        exec.execute(new ExceptionThread2());
    }
} /* Output: (90% match)
HandlerThreadFactory@de6ced creating new Thread
created Thread[Thread-0,5,main]
eh = MyUncaughtExceptionHandler@1fb8ee3
run() by Thread[Thread-0,5,main]
```



```
eh = MyUncaughtExceptionHandler@1fb8ee3
caught java.lang.RuntimeException
*///:~
```

In questo codice è stato aggiunto un meccanismo di tracciamento per verificare che ai thread creati dalla factory sia correttamente assegnato il nuovo **UncaughtExceptionHandler**. Come potete vedere, ora le eccezioni non intercettate vengono gestite da **uncaughtException**.

L'esempio precedente consente di impostare il gestore caso per caso. Se sape-te di poter utilizzare lo stesso gestore di eccezioni in ogni punto della vostra applicazione, una tecnica ancora più semplice consiste nell'implementazio-ne di un “gestore di eccezioni non intercettate” predefinito, che imposta un campo **static** all'interno della classe **Thread**.

```
//: concurrency/SettingDefaultHandler.java
import java.util.concurrent.*;

public class SettingDefaultHandler {
    public static void main(String[] args) {
        Thread.setDefaultUncaughtExceptionHandler(
            new MyUncaughtExceptionHandler());
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new ExceptionThread());
    }
} /* Output:
caught java.lang.RuntimeException
*///:~
```

Il gestore **defaultUncaughtExceptionHandler** viene chiamato soltanto se non è stato previsto un gestore di eccezioni non intercettate per ogni thread. Il siste-ma verifica se esiste una versione di quest'ultimo gestore, e qualora non la trovi controlla se il gruppo di thread specializza il proprio metodo **uncaughtExcep-tion()**; in caso contrario, chiama **defaultUncaughtExceptionHandler**.

## Condivisione delle risorse

Potete considerare un programma a thread singolo (*single-threaded*) come un'unica entità che si muove nell'ambito dello spazio del problema ed esegue una sola operazione per volta. Trattandosi di una sola entità non dovrete mai



occuparvi del problema di entità multiple che cercano di utilizzare la stessa risorsa nello stesso tempo: persone che provano a parcheggiare nello stesso spazio, che cercano di transitare da una porta nello stesso momento o di parlare simultaneamente. Con la concorrenza, tuttavia, le entità non sono più singole ed esiste effettivamente la possibilità che due o più task interferiscano a vicenda: se non fate in modo di prevenire tali scontri potreste ritrovarvi con vari task che cercano di accedere al medesimo conto corrente bancario nello stesso momento, di stampare con la stessa stampante, di registrare la stessa valvola e così via.

### Accesso *improprio* alle risorse

Considerate l'esempio seguente, relativo a un task che genera numeri pari poi "consumati" da altri task; in questo caso, l'unico compito dei task consumatori è controllare la validità dei numeri pari.

In primo luogo definite **EvenChecker**, il task consumatore, poiché sarà riutilizzato negli esempi successivi. Per dissociare **EvenChecker** dai diversi tipi di generatori che sperimenterete, create una classe astratta chiamata **IntGenerator**, contenente i requisiti indispensabili a **EvenChecker**, vale a dire la disponibilità di un metodo **next()** e la possibilità di annullare il task, fornita dal metodo **cancel()**. Questa classe non implementa l'interfaccia **Generator** poiché deve produrre un **int**, e i generici non supportano i tipi primitivi come parametri.

```
//: concurrency/IntGenerator.java

public abstract class IntGenerator {
    private volatile boolean canceled = false;
    public abstract int next();
    // Permette la cancellazione di questo task:
    public void cancel() { canceled = true; }
    public boolean isCanceled() { return canceled; }
} ///:-~
```

La classe **IntGenerator** ha un metodo **cancel()** per cambiare lo stato di un flag **boolean canceled**, e un metodo **isCanceled()** per verificare se l'oggetto è stato annullato. Essendo di tipo **boolean**, il flag **canceled** è *atomico*: questo significa che semplici operazioni quali l'assegnazione e il ritorno di un valore non possono essere interrotte, dal momento che il flag non può trovarsi in uno stato diverso da vero o falso. Il flag **canceled** è anche **volatile**, per garantirne la visibilità. Avrete occasione di analizzare meglio i concetti di atomicità e di visibilità nel proseguito del capitolo.



Qualsiasi classe **IntGenerator** può essere testata con la seguente classe **EvenChecker**.

```
//: concurrency/EvenChecker.java
import java.util.concurrent.*;

public class EvenChecker implements Runnable {
    private IntGenerator generator;
    private final int id;
    public EvenChecker(IntGenerator g, int ident) {
        generator = g;
        id = ident;
    }
    public void run() {
        while(!generator.isCancelled()) {
            int val = generator.next();
            if(val % 2 != 0) {
                System.out.println(val + " not even!");
                generator.cancel(); // Cancella tutti gli EvenCheckers
            }
        }
    }
    // Verifica qualsiasi tipo di IntGenerator:
    public static void test(IntGenerator gp, int count) {
        System.out.println("Press Control-C to exit");
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < count; i++)
            exec.execute(new EvenChecker(gp, i));
        exec.shutdown();
    }
    // Valore predefinito per count:
    public static void test(IntGenerator gp) {
        test(gp, 10);
    }
} ///:~
```

Notate che in questo esempio la classe che può essere annullata non è **Runnable**. Invece, come potete vedere in **run()**, tutti i task di **EvenChecker** che dipendono



dall'oggetto **IntGenerator** lo testano per vedere se è stato annullato. In questo modo, i task che condividono la risorsa comune (**IntGenerator**) monitorano il segnale di completamento per quella risorsa. Questo approccio impedisce il verificarsi della cosiddetta corsa critica (*race condition*), nella quale due o più task "si precipitano" a far fronte a una condizione, finendo per scontrarsi o produrre risultati incoerenti. Dovete tenere in considerazione e cauterlarvi da tutte le eventualità che potrebbero impedire il funzionamento corretto di un sistema concorrente: per esempio, un task non dovrebbe dipendere da un altro task, poiché l'ordine di completamento dei task non è assicurato. In questo caso specifico, facendo in modo che la creazione dei task dipenda da un oggetto non task, eliminate la potenziale condizione di corsa critica.

Il metodo **test()** imposta ed esegue un test per qualunque tipo di **IntGenerator**, avviando alcuni **EvenChecker** che utilizzano lo stesso **IntGenerator**. Se l'**IntGenerator** dovesse causare un errore, **test()** lo segnalerà e ritornerà; in caso contrario dovrete digitare Ctrl+C per terminarlo.

I task di **EvenChecker** leggono e verificano costantemente i valori degli **IntGenerator** associati. Tenete presente che se **generator.isCanceled()** è **true**, **run()** ritornerà, indicando all'**Executor** in **EvenChecker.test()** che il task è terminato. Qualsiasi task di **EvenChecker** può chiamare **cancel()** a fronte dell'**IntGenerator** a esso collegato, il che provocherà l'interruzione "non brusca" (*graceful shutdown*) di tutti gli **EvenChecker** che utilizzano lo stesso **IntGenerator**. Nei prossimi paragrafi vedrete che Java offre meccanismi più generali per terminare i thread.

Il primo **IntGenerator** da analizzare ha un metodo **next()** che produce una serie di numeri pari.

```
//: concurrency/EvenGenerator.java
// Collisione di thread.

public class EvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    public int next() {
        ++currentEvenValue; // Questo e' un punto pericoloso!
        ++currentEvenValue;
        return currentEvenValue;
    }
    public static void main(String[] args) {
        EvenChecker.test(new EvenGenerator());
    }
}
```



```
 } /* Output: (Esempio)
Press Control-C to exit
89476993 not even!
89476993 not even!
*///:~
```

Come potete vedere a fronte del commento “Questo è un punto pericoloso”, è possibile che un task chiami `next()` dopo che un’altra operazione ha eseguito il primo incremento di `currentEvenValue`, ma non il secondo, il che metterebbe il valore in una condizione “errata”. Per dimostrare che questo evento può verificarsi, `EvenChecker.test()` crea un gruppo di oggetti `EvenChecker` per leggere continuamente l’output di `EvenGenerator` e controllare costantemente che il risultato sia pari: se non è così, l’errore sarà segnalato e il programma terminerà.

Alla fine, questo programma finirà per interrompersi poiché i task di `EvenChecker` possono accedere alle informazioni in `EvenGenerator` mentre questo è in una condizione “errata”. Tuttavia, il problema potrebbe non essere rilevato prima che `EvenGenerator` abbia completato molte iterazioni: questo dipende, naturalmente, dalle particolarità del sistema operativo e da altri dettagli implementativi. Se volete che il programma si interrompa in un tempo più breve, provate a inserire una chiamata a `yield()` tra il primo e il secondo incremento. Questa è una dimostrazione dei problemi che possono affliggere le applicazioni multithread: se il rischio di interruzione del programma è molto ridotto, tutto potrebbe sembrare corretto anche se effettivamente esiste un malfunzionamento.

È importante notare che l’operazione di incremento in sé richiede diversi passaggi, e che il task può essere sospeso dal meccanismo di threading nel mezzo di un incremento; questo significa che in Java l’incremento non è un’operazione atomica, pertanto anche un solo incremento non può essere considerato sicuro a meno che il task non venga protetto.

### ***Risoluzione di conflitti tra risorse condivise***

L’esempio precedente evidenzia una caratteristica fondamentale nell’utilizzo dei thread: non si può mai sapere quando un thread potrebbe funzionare. Immaginatevi seduti a tavola con una forchetta in mano, in attesa di servirvi l’ultima fetta di arrosto da un vassoio, ma ecco che quando state per prenderla all’istante la fetta scompare: il vostro thread è stato improvvisamente sospeso e un altro commensale è entrato e ha terminato l’arrosto al vostro posto. Questo è il problema con cui avete a che fare quando scrivete programmi concorrenti. Affinché la concorrenza funzioni dovete ricorrere a tecniche tali



da impedire che due task accedano alla stessa risorsa, quantomeno durante le fasi elaborative più critiche.

Impedire questo genere di conflitti non richiede altro che impostare un blocco (*lock*) su una risorsa, nel momento in cui viene utilizzata da un task. Il primo task che accede a una risorsa la blocca, in modo che gli altri task non possano accedervi finché la risorsa non viene sbloccata; a quel punto un altro task bloccherà la risorsa per utilizzarla e così via.

Per risolvere il problema della collisione tra i thread, praticamente ogni schema di concorrenza serializza l'accesso alle risorse condivise, facendo in modo che soltanto un task per volta acceda alla risorsa. Questo comportamento si ottiene incapsulando la porzione di codice desiderata in un'istruzione che permetta a una sola operazione per volta di "transitare" da quella porzione di codice. Poiché questa istruzione produce un effetto di esclusione reciproca (*mutual exclusion*) un termine comunemente utilizzato per questo meccanismo è *mutex*.

Immaginate, davanti a una cabina telefonica, un gruppo di persone (task guidati da thread) ognuna delle quali abbia bisogno di utilizzare in modo esclusivo il telefono pubblico (la risorsa comune). Quando una persona occupa la cabina, tutti gli altri task si troveranno "bloccati" e aspetteranno il loro turno per accedere al servizio.

L'analogia si discosta dalla realtà quando la cabina torna libera ed è tempo di garantire l'accesso a un altro task. Non vi è una fila ordinata di persone e non è possibile sapere con certezza chi sarà il prossimo ad accedere alla cabina, poiché il thread scheduler non arriva a un tale livello di determinismo. Invece, è come se davanti alla cabina vi fosse un gruppo di task bloccati, e non appena il task che ha occupato la cabina la lascia libera, ecco che il task in quel momento più vicino vi accede, bloccandola di nuovo. Come si è detto, è possibile influenzare il comportamento del pianificatore di thread tramite `yield()` e `setPriority()`, tuttavia gli effetti possono essere molto diversi, a seconda della piattaforma e della JVM implementata.

Come supporto per impedire i conflitti di risorse, Java offre la parola chiave **synchronized**; quando un task vuole eseguire una parte di codice soggetta a questa parola chiave, controlla per vedere se il blocco è attivo, lo acquisisce, esegue il codice e lo rimette a disposizione.

Generalmente la risorsa comune è una porzione di memoria sotto forma di oggetto, ma può anche essere un file, una porta I/O o un dispositivo esterno, quale una stampante. Per gestire l'accesso a una risorsa comune dovete per prima cosa inserirla in un oggetto. A questo punto qualunque metodo utilizzi la risorsa può essere reso **synchronized**. Se un task sta chiamando uno dei metodi **synchronized**, a tutti gli altri sarà impedito l'accesso a qualsiasi



metodo **synchronized** di quell'oggetto, fino a quando il primo task non ritorna dalla sua chiamata.

Nel codice di produzione, come avete visto, dovreste rendere **private** gli elementi di dati di una classe e accedere a quella porzione di memoria soltanto per mezzo di metodi. Potete impedire eventuali conflitti dichiarando quei metodi **synchronized**, come nell'esempio seguente:

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

Tutti gli oggetti contengono automaticamente un solo lock, chiamato anche *monitor lock*. Quando chiamate un metodo **synchronized** l'oggetto viene bloccato e nessun altro metodo **synchronized** dello stesso oggetto può essere chiamato finché il primo non termina e rilascia il lock. Per quanto riguarda i metodi dell'esempio precedente, se a **f()** viene richiesto un oggetto da un task, un altro task non potrà richiedere lo stesso oggetto chiamando **f()** o **g()**, fino a quando **f()** non abbia completato l'esecuzione e rilasciato il lock. Questo dimostra che esiste un solo lock che viene ripartito tra tutti i metodi **synchronized** di un determinato oggetto, e che tale lock può essere utilizzato per impedire la scrittura della memoria dell'oggetto da parte di più task contemporaneamente.

Ricordate che è molto importante rendere i campi **private** quando lavorate con la concorrenza, altrimenti la parola chiave **synchronized** non potrà impedire che un altro task acceda al campo direttamente, provocando così un conflitto.

Un task potrebbe acquisire più volte il lock su un oggetto: questo accade se un metodo chiama un secondo metodo sullo stesso oggetto, che a sua volta chiama un altro metodo sullo stesso oggetto e così via. La JVM tiene traccia del numero di volte che l'oggetto è stato bloccato: non appena l'oggetto viene sbloccato il conteggio riparte da zero. Quando un task acquisisce per la prima volta il lock, il conteggio vale uno, e viene incrementato ogni volta che lo stesso task acquisisce un altro lock sullo stesso oggetto. Naturalmente l'acquisizione di lock multipli è concessa soltanto al task che lo ha acquisito la prima volta. Ogni volta che il task esce da un metodo **synchronized** il conteggio diminuisce, fino ad arrivare a zero, liberando così il lock a favore di altri task.

Esiste anche un solo lock per classe, come componente dell'oggetto **Class** della classe; per mezzo di questo lock i metodi **static synchronized** possono bloccarsi a vicenda impedendo l'accesso simultaneo ai dati **static** a livello di classe.

Dovreste applicare la sincronizzazione attenendovi a quella che viene chiamata *Regola di sincronizzazione di Brian*, tratta da *Java Concurrency in Practice*, di Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes e Doug Lea (Addison-Wesley, 2006).



*"Se state scrivendo una variabile che potrebbe essere poi letta da un altro thread, o leggendo una variabile che potrebbe essere stata scritta per ultima da un altro thread, non soltanto dovete utilizzare la sincronizzazione, ma sia il thread-lettore sia il thread-scrittore dovranno sincronizzarsi tramite lo stesso monitor lock".*

Se la vostra classe ha più metodi che gestiscono dati critici, dovete sincronizzare tutti i metodi. In caso contrario i metodi non sincronizzati sarebbero liberi di ignorare il lock dell'oggetto e potrebbero essere chiamati senza alcun controllo. Questo è un punto importante: ogni metodo che accede a una risorsa comune critica deve essere di tipo **synchronized**, oppure non funzionerà correttamente.

### Sincronizzazione di EvenGenerator

Aggiungendo **synchronized** a **EvenGenerator.java** potete impedirne l'accesso da parte di altri thread.

```
//: concurrency/SynchronizedEvenGenerator.java
// Semplifica le esclusioni reciproche (mutex) mediante la
// parola chiave synchronized.
// {RunByHand}

public class
SynchronizedEvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    public synchronized int next() {
        ++currentEvenValue;
        Thread.yield(); // Anticipa l'interruzione
        ++currentEvenValue;
        return currentEvenValue;
    }
    public static void main(String[] args) {
        EvenChecker.test(new SynchronizedEvenGenerator());
    }
} ///:~
```

Tra le due istruzioni di incremento viene inserita una chiamata a **Thread.yield()**, per aumentare la probabilità di un cambio di contesto mentre **currentEvenValue** si trova nello stato di dispari. Poiché la funzionalità di esclusione reciproca (*mutex*) impedisce la presenza di più di un'operazione per



volta nella sezione critica, il cambio di contesto non produrrà un'interruzione; in ogni caso, la chiamata a `yield()` è una tecnica utile per "incoraggiare" un errore che dovrebbe comunque verificarsi.

La prima operazione che accede a `next()` acquisisce il lock, e qualsiasi altro task che cercasse di acquisire il lock ne sarà impedito finché il primo task non abbia rilasciato il lock; a quel punto il pianificatore (*task scheduler*) selezionerà un altro task in attesa. Grazie a questo meccanismo, soltanto un'operazione per volta potrà transitare dal codice che è sottoposto al controllo del mutex.

**Esercizio 11** (3) Create una classe contenente due campi di dati e un metodo che gestisce tali campi in un processo a più fasi, in modo che durante l'esecuzione di quel metodo i campi si trovino in una "condizione impropria", variabile secondo i criteri che definirete. Aggiungete metodi per la lettura dei campi, e create thread multipli che chiamino i vari metodi e dimostrino che i dati sono visibili nella loro "condizione impropria". Infine, risolvete il problema con la parola chiave `synchronized`.

### Utilizzo di oggetti Lock esplicativi

La libreria `java.util.concurrent` di Java SE5 contiene anche un meccanismo di mutex esplicito, definito in `java.util.concurrent.locks`. L'oggetto `Lock` deve essere intenzionalmente creato, bloccato e sbloccato, di conseguenza produce codice meno elegante rispetto alla forma nativa, pur essendo più flessibile nella risoluzione di determinati problemi. Quella che segue è la versione di `SynchronizedEvenGenerator.java` riscritta utilizzando i `Lock` esplicativi.

```
//: concurrency/MutexEvenGenerator.java
// Impedisce il conflitto di thread con i mutex.
// {RunByHand}
import java.util.concurrent.locks.*;

public class MutexEvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    private Lock lock = new ReentrantLock();
    public int next() {
        lock.lock();
        try {
            ++currentEvenValue;
            Thread.yield(); // Anticipa l'interruzione
            ++currentEvenValue;
            return currentEvenValue;
        }
    }
}
```



```
    } finally {
        lock.unlock();
    }
}
public static void main(String[] args) {
    EvenChecker.test(new MutexEvenGenerator());
}
} ///:-~
```

La classe **MutexEvenGenerator** aggiunge un mutex chiamato **lock** e si serve dei metodi **lock()** e **unlock()** per creare una sezione critica all'interno di **next()**. Utilizzando gli oggetti **Lock**, è importante interiorizzare la forma idiomatica seguente: subito dopo la chiamata a **lock()** dovete inserire un'istruzione **try-finally** con **unlock()** nella clausola **finally**; infatti, questo è l'unico modo per garantire che il lock sarà sempre rilasciato. Notate che l'istruzione **return** deve trovarsi all'interno della clausola **try** per assicurarsi che l'**unlock()** non accada troppo presto, esponendo i dati a un secondo task.

Nonostante la tecnica **try-finally** richieda uno quantità maggiore di codice rispetto all'utilizzo della parola chiave **synchronized**, rappresenta uno dei vantaggi degli oggetti **Lock** esplicativi. Se qualcosa non funziona con la parola chiave **synchronized** verrà gestita un'eccezione, ma non avrete modo di sotoporre a cleanup il vostro sistema per mantenerlo in buone condizioni; di contro, con gli oggetti **Lock** esplicativi potete mantenere la condizione adeguata mediante l'istruzione **finally**.

In generale, se ricorrete a **synchronized** avrete una quantità inferiore di codice da scrivere e l'opportunità di errori da parte dell'utente programmatore risulterà notevolmente ridotta; pertanto utilizzerete gli oggetti **Lock** esplicativi soltanto per risolvere particolari problemi. Per esempio, la parola chiave **synchronized** non consente di acquisire un lock andando per tentativi, né di provare ad acquisirlo per un determinato tempo e poi rinunciare all'operazione. A questo scopo dovete servirvi della libreria **concurrent**.

```
//: concurrency/AttemptLocking.java
// I Lock nella libreria di concorrenza permettono di
// sospendere i tentativi di acquisizione di un lock.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AttemptLocking {
    private ReentrantLock lock = new ReentrantLock();
```



```
public void untimed() {
    boolean captured = lock.tryLock();
    try {
        System.out.println("tryLock(): " + captured);
    } finally {
        if(captured)
            lock.unlock();
    }
}
public void timed() {
    boolean captured = false;
    try {
        captured = lock.tryLock(2, TimeUnit.SECONDS);
    } catch(InterruptedException e) {
        throw new RuntimeException(e);
    }
    try {
        System.out.println("tryLock(2, TimeUnit.SECONDS): " +
                           captured);
    } finally {
        if(captured)
            lock.unlock();
    }
}
public static void main(String[] args) {
    final AttemptLocking al = new AttemptLocking();
    al.untimed(); // True - il lock e' disponibile
    al.timed(); // True - il lock e' disponibile
    // Ora crea un task separato per intercettare il lock:
    new Thread() {
        { setDaemon(true); }
        public void run() {
            al.lock.lock();
            System.out.println("acquired");
        }
    }.start();
    Thread.yield(); // Offre una possibilita' al secondo task
}
```



```
    al.untimed(); // False - lock intercettato da un task
    al.timed();   // False - lock intercettato da un task
}
} /* Output:
tryLock(): true
tryLock(2, TimeUnit.SECONDS): true
acquired
tryLock(): false
tryLock(2, TimeUnit.SECONDS): false
*/://:-
```

Un **ReentrantLock** consente di cercare di acquisire il lock per tentativi, in modo che, se questo fosse già in possesso di qualche altro task, possiate decidere di rinunciare ed eseguire un'altra operazione, anziché attendere il rilascio del lock, come potete vedere nel metodo **untimed()**. In **timed()** viene effettuato un tentativo per acquisire il lock, che può interrompersi dopo 2 secondi: notate l'utilizzo della classe **TimeUnit** di Java SE5 che specifica le unità di tempo. In **main()** viene creato un **Thread** separato come classe anonima, che acquisisce il lock in modo che i metodi **untimed** e **timed()** abbiano qualcosa da contendersi.

L'oggetto **Lock** esplicito assicura anche un controllo più efficace sul bloccaggio e lo sbloccaggio rispetto al lock **synchronized** nativo, utile per implementare strutture di sincronizzazione speciali: in particolare il cosiddetto *hand-over-hand locking o lock coupling*, utilizzato per percorrere i nodi di un elenco collegato; il codice che gestisce l'attraversamento deve intercettare il lock del nodo successivo prima che possa essere rilasciato il lock del nodo corrente.

## Atomicità e volatilità

Una convinzione errata che emerge spesso nelle discussioni sui thread Java è che le operazioni atomiche non debbano essere sincronizzate. Si definisce “operazione atomica” quella che non può essere interrotta dal pianificatore di thread: quando l’operazione inizia, giunge a completamento prima che esista la possibilità di un cambio di contesto. Affidarsi all’atomicità è un approccio ingannevole e pericoloso: dovreste provare a utilizzare l’atomicità, in luogo della sincronizzazione, soltanto se avete esperienza nelle tecniche di concorrenza o se siete supportati da un esperto. Per valutare la vostra capacità di giocare con questo “tipo di fuoco”, considerate il test seguente.



*"Il test di Goetz: soltanto se siete all'altezza di scrivere una JVM che offre elevate prestazioni su una moderna CPU, disporrete delle competenze indispensabili per ritenere di poter evitare la sincronizzazione!"<sup>8</sup>*

È utile conoscere l'atomicità e sapere che, con altre tecniche avanzate, è stata adottata per implementare alcuni dei componenti più "intelligenti" della libreria **java.util.concurrent**. Evitate tuttavia di farvi affidamento: in proposito, ricordate la regola di sincronizzazione dello stesso autore, di cui si è parlato nel paragrafo "Risoluzione di conflitti tra risorse condivise".

L'atomicità si applica a "operazioni semplici" sui tipi primitivi, tranne **long** e **double**. La lettura e la scrittura di variabili primitive diverse da **long** e **double** sono garantite come accesso alla memoria, in entrata e uscita, sotto forma di operazioni atomiche (indivisibili). Tuttavia, la JVM esegue la lettura e la scrittura di segmenti a 64 bit (variabili **long** e **double**) come due operazioni separate di segmenti di 32 bit, con il rischio che possa verificarsi un cambio di contesto nel corso di un'operazione di lettura o di scrittura. In questo modo, task diversi potrebbero accedere a risultati errati: un fenomeno chiamato talvolta *word tearing*, che fa sì che un valore sia visibile anche se modificato soltanto parzialmente. La condizione di atomicità è tuttavia ottenibile per semplici assegnazioni e restituzioni mediante la parola chiave **volatile**, in fase di definizione di una variabile **long** o **double**: ricordate che **volatile** funziona in modo corretto soltanto a partire da Java SE5. Differenti JVM potrebbero offrire garanzie più solide, in ogni caso non dovreste basarvi su caratteristiche specifiche di una determinata piattaforma.

Quindi, le operazioni atomiche non possono essere interrotte dal meccanismo di threading. I programmati esperti possono trarre beneficio da questa caratteristica per scrivere codice *lock-free*, che non richiede sincronizzazione, ma persino questa è una semplificazione eccessiva. A volte, anche quando sembra che un'operazione atomica offra garanzie di sicurezza potrebbe non essere così. La maggior parte dei lettori di questo manuale non è in grado di superare il test di Goetz, quindi non sarà in grado di rimpiazzare la sincronizzazione con operazioni atomiche. In genere, tentare di rimuovere la sincronizzazione è indice di ottimizzazione prematura e può dare luogo a numerosi problemi, fornendo peraltro scarsi vantaggi, se non addirittura nessuno.

Diversamente da quanto avviene sui sistemi monoprocessoressi, sui sistemi multiprocessore, che stanno via via diffondendosi sotto forma di processori *multicore* (CPU multiple su un unico circuito integrato), la visibilità è più proble-

---

8. Questa considerazione è stata suggerita all'autore da alcuni commenti scherzosi di Brian Goetz, più volte citato in questo volume, un esperto di concorrenza che ha collaborato alla stesura di questo capitolo.



matica rispetto all'atomicità. I cambiamenti operati da un task, anche se sono atomici nel senso che non sono interrompibili, potrebbero non essere visibili ad altri task, per esempio perché le modifiche potrebbero essere temporaneamente memorizzate nella memoria cache del processore locale: pertanto, diversi task avranno una visione differente dello stato dell'applicazione. Il meccanismo di sincronizzazione, d'altro canto, impone che le modifiche eseguite da un task su un sistema multiprocessore siano disponibili all'intera applicazione. Senza la sincronizzazione non è possibile prevedere quando tali cambiamenti diventeranno visibili. Anche la parola chiave **volatile** assicura la visibilità nell'ambito dell'applicazione. Se dichiarate un campo **volatile**, non appena viene eseguita un'operazione di scrittura su questo campo tutte le letture vedranno il cambiamento. Questo è vero anche se sono coinvolte memorie cache locali: i campi **volatile** vengono scritti immediatamente sulla memoria principale, e le letture sono eseguite da quest'ultima.

È importante comprendere che atomicità e volatilità sono concetti distinti. Un'operazione atomica su un campo non **volatile** non sarà necessariamente riportata nella memoria principale, di conseguenza un altro task che legga questo campo potrebbe non vedere il nuovo valore. Se vari task hanno accesso allo stesso campo, questo dovrebbe essere **volatile**; in caso contrario dovrebbe essere raggiungibile soltanto tramite la sincronizzazione. La sincronizzazione provoca anche lo svuotamento, con contestuale scrittura, della memoria principale (il cosiddetto *flushing*): pertanto, se un campo è interamente protetto da metodi o blocchi **synchronized** non sarà necessario renderlo **volatile**.

Ogni scrittura eseguita da un task sarà visibile a quest'ultimo, quindi il campo non deve essere reso **volatile** se è visibile soltanto all'interno di un task.

La parola chiave **volatile** non ha effetto su un campo il cui valore attuale dipende dal valore precedente, come avviene per gli incrementi di un contatore, né su campi i cui valori sono vincolati a quelli di altri campi, come i limiti superiore (**upper**) e inferiore (**lower**) di una classe **Range** che deve sottostare al vincolo **lower <= upper**. Di solito, l'utilizzo di **volatile** in luogo della parola chiave **synchronized** può essere considerato sicuro soltanto se la classe ha un solo campo modificabile. Anche in questo caso la vostra prima scelta dovrebbe cadere sulla parola chiave **synchronized**: è l'approccio più sicuro, e provare ad adottare altre soluzioni è rischioso.

Che cosa qualifica un'operazione come atomica? L'assegnazione e la restituzione del valore di un campo generalmente saranno atomici. Tuttavia, in C++ anche le seguenti operazioni potrebbero essere atomiche:

```
i++;      // Potrebbe essere atomico in C++
i += 2; // Potrebbe essere atomico in C++
```



Ma in C++ questo dipende dal compilatore e dal processore. In questo linguaggio non potete scrivere codice multi-piattaforma basato sull'atomicità, poiché C++ non ha un *modello di memoria* costante come Java SE5.<sup>9</sup>

In Java le suddette operazioni non sono atomiche, come potete vedere dalle istruzioni JVM prodotte dai seguenti metodi.

```
//: concurrency/Atomicity.java
// {Exec: javap -c Atomicity}

public class Atomicity {
    int i;
    void f1() { i++; }
    void f2() { i += 3; }
} /* Output: (Esempio)
...
void f1();
Code:
  0:   aload_0
  1:   dup
  2:   getfield      #2; //Field i:I
  5:   iconst_1
  6:   iadd
  7:   putfield      #2; //Field i:I
 10:  return

void f2();
Code:
  0:   aload_0
  1:   dup
  2:   getfield      #2; //Field i:I
  5:   iconst_3
  6:   iadd
  7:   putfield      #2; //Field i:I
 10:  return
*///:-
```

9. A questo dovrebbe rimediare lo standard di Ecma 372 C++/CLI ([www.ecma-international.org/publications/standards/Ecma-372.htm](http://www.ecma-international.org/publications/standards/Ecma-372.htm)).



Ogni istruzione produce le istruzioni "get" e "put", intervallate da altre istruzioni. Tra get e put un'altra operazione potrebbe modificare il campo: di conseguenza le operazioni non sono atomiche.

Se applicate passivamente l'idea dell'atomicità, vedrete che nel programma seguente il metodo `getValue()` si adatta alla sua descrizione.

```
//: concurrency/AtomicityTest.java
import java.util.concurrent.*;

public class AtomicityTest implements Runnable {
    private int i = 0;
    public int getValue() { return i; }
    private synchronized void evenIncrement() { i++; i++; }
    public void run() {
        while(true)
            evenIncrement();
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        AtomicityTest at = new AtomicityTest();
        exec.execute(at);
        while(true) {
            int val = at.getValue();
            if(val % 2 != 0) {
                System.out.println(val);
                System.exit(0);
            }
        }
    }
} /* Output: (Esempio)
191583767
*///:~
```

Tuttavia il programma troverà valori dispari e si concluderà. Sebbene `return i` sia effettivamente un'operazione atomica, la mancanza di sincronizzazione fa sì che il valore possa essere letto mentre l'oggetto si trova in uno stato intermedio instabile. Oltre a questo, dal momento che `i` è anche non `volatile`, ci saranno problemi di visibilità. Entrambi i metodi `getValue()` ed `evenIn-`



ment() devono essere **synchronized**. Soltanto gli esperti di concorrenza sono qualificati per eseguire ottimizzazioni in situazioni come questa: di nuovo, dovreste applicare la Regola di sincronizzazione di Brian.

Come secondo esempio considerate qualcosa di più semplice: una classe che produce numeri di serie.<sup>10</sup>

Ogni volta che viene chiamato, il metodo **nextSerialNumber()** deve restituire un valore univoco al chiamante.

```
//: concurrency/SerialNumberGenerator.java

public class SerialNumberGenerator {
    private static volatile int serialNumber = 0;
    public static int nextSerialNumber() {
        return serialNumber++; // Non sicuro a livello di thread
    }
} //:~
```

**SerialNumberGenerator** è probabilmente la classe più semplice che possiate immaginare; se provenite da un'esperienza di C++ o di un altro linguaggio di basso livello potrete aspettarvi che l'incremento sia un'operazione atomica, poiché in C++ spesso è implementato come istruzione del microprocessore, sebbene non in modo affidabile e indipendente dalla piattaforma.

Come si è notato in precedenza, tuttavia, in Java un incremento non è atomico e coinvolge una lettura e una scrittura: questo lascia spazio a problemi di threading anche in un'operazione così semplice. Come vedrete, in questo caso il problema non è tanto la volatilità quanto il fatto che il metodo **nextSerialNumber()** accede a un valore condiviso e modificabile senza sincronizzazione.

Il campo **serialNumber** è **volatile** poiché ogni thread potrebbe avere uno stack locale in cui conservare le copie di alcune variabili. Definendo una variabile come **volatile**, questa parola chiave indica al compilatore di non eseguire alcuna ottimizzazione che potrebbe rimuovere le operazioni di lettura e scrittura in sincronizzazione esatta con i dati locali nei thread. In effetti la lettura e la scrittura operano direttamente sulla memoria e non sono poste in cache. Inoltre, la parola chiave **volatile** obbliga il compilatore a riordinare gli accessi durante l'ottimizzazione. Comunque sia, **volatile** non ha effetto sul fatto che un incremento non è un'operazione atomica.

---

10. Questa classe è stata ispirata da *Effective Java Programming Language Guide* di Joshua Bloch (Addison-Wesley, 2001), pag. 190.



In generale dovreste rendere **volatile** un campo cui possono accedere simultaneamente più task, se almeno uno di questi accessi è in scrittura. Per esempio, un campo utilizzato come flag per interrompere un task deve essere **volatile** per evitare che possa essere messo temporaneamente in cache: in tal caso, infatti, al momento di eseguire modifiche al flag dall'esterno del task il valore in cache non verrebbe modificato e il task non saprebbe di doversi interrompere.

Per testare **SerialNumberGenerator** è necessario un insieme che non esaurisca la memoria, poiché in questo caso occorrerebbe molto tempo per rilevare il problema. La classe **CircularSet** mostrata nell'esempio seguente riutilizza la memoria utilizzata per memorizzare gli **int**, dando per scontato che la possibilità di collisione con i valori sovrascritti sia minima nel momento in cui sposterete gli **int**. I metodi **add()** e **contains()** sono **synchronized** per impedire conflitti di thread.

```
//: concurrency/SerialNumberChecker.java
// Operazioni che possono sembrare sicure non lo sono,
// in presenza di thread.
// {Args: 4}
import java.util.concurrent.*;

// Riutilizza lo spazio di archiviazione per non esaurire
// la memoria:
class CircularSet {
    private int[] array;
    private int len;
    private int index = 0;
    public CircularSet(int size) {
        array = new int[size];
        len = size;
        // Inizializza a un valore non prodotto da
        // SerialNumberGenerator:
        for(int i = 0; i < size; i++)
            array[i] = -1;
    }
    public synchronized void add(int i) {
        array[index] = i;
        // Ingloba l'indice e sovrascrive i vecchi elementi:
        index = ++index % len;
    }
}
```



```
public synchronized boolean contains(int val) {
    for(int i = 0; i < len; i++)
        if(array[i] == val) return true;
    return false;
}

public class SerialNumberChecker {
    private static final int SIZE = 10;
    private static CircularSet serials =
        new CircularSet(1000);
    private static ExecutorService exec =
        Executors.newCachedThreadPool();
    static class SerialChecker implements Runnable {
        public void run() {
            while(true) {
                int serial =
                    SerialNumberGenerator.nextSerialNumber();
                if(serials.contains(serial)) {
                    System.out.println("Duplicate: " + serial);
                    System.exit(0);
                }
                serials.add(serial);
            }
        }
    }
    public static void main(String[] args) throws Exception {
        for(int i = 0; i < SIZE; i++)
            exec.execute(new SerialChecker());
        // Interrompe dopo n secondi se è fornito un argomento:
        if(args.length > 0) {
            TimeUnit.SECONDS.sleep(new Integer(args[0]));
            System.out.println("No duplicates detected");
            System.exit(0);
        }
    }
}
```



```
} /* Output: (Esempio)
Duplicate: 8468656
*/://:~
```

**SerialNumberChecker** ha un oggetto static **CircularSet** contenente tutti i numeri di serie che sono stati generati, e una classe nidificata **SerialChecker** che si accerta che tali valori siano univoci. Creando task multipli che si contendono i numeri di serie scoprirete che, se lasciate in esecuzione il programma per un tempo sufficiente, alla fine i task possono ottenere un duplicato. Per risolvere il problema dovete aggiungere la parola chiave **synchronized** al metodo **nextSerialNumber()**.

Le operazioni atomiche che si suppone siano sicure sono la lettura e l'assegnazione di tipi primitivi. Come avete visto in **AtomicityTest.java**, però, è ancora possibile che un'operazione atomica acceda al vostro oggetto mentre si trova in uno stato intermedio instabile. Presupporre che una certa operazione sia atomica è dunque ingannevole e pericoloso: l'approccio più cauto è attenersi alla Regola sulla sincronizzazione di Brian.

**Esercizio 12** (3) Correggete **AtomicityTest.java** utilizzando la parola chiave **synchronized**. Potete dimostrarne il corretto funzionamento?

**Esercizio 13** (1) Correggete **SerialNumberChecker.java** utilizzando la parola chiave **synchronized**. Potete dimostrarne il corretto funzionamento?

### Classi atomiche

Java SE5 offre speciali classi atomiche variabili, quali **AtomicInteger**, **AtomicLong**, **AtomicReference** ecc., che garantiscono operazioni di aggiornamento condizionali atomiche nel formato:

```
boolean compareAndSet(expectedValue, updateValue);
```

Queste classi sono state ideate per la messa a punto dell'atomicità a livello macchina che è caratteristica di alcuni moderni processori, di conseguenza di norma non avrete bisogno di servirvene. In alcune occasioni, però, tali classi possono risultare pratiche anche nella normale programmazione, quando sia necessario ottimizzare le prestazioni. Per esempio potete riscrivere **AtomicityTest.java** per utilizzare **AtomicInteger**.

```
//: concurrency/AtomicIntegerTest.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;
```



```
public class AtomicIntegerTest implements Runnable {
    private AtomicInteger i = new AtomicInteger(0);
    public int getValue() { return i.get(); }
    private void evenIncrement() { i.addAndGet(2); }
    public void run() {
        while(true)
            evenIncrement();
    }
    public static void main(String[] args) {
        new Timer().schedule(new TimerTask() {
            public void run() {
                System.err.println("Aborting");
                System.exit(0);
            }
        }, 5000); // Il programma termina dopo 5 secondi
        ExecutorService exec = Executors.newCachedThreadPool();
        AtomicIntegerTest ait = new AtomicIntegerTest();
        exec.execute(ait);
        while(true) {
            int val = ait.getValue();
            if(val % 2 != 0) {
                System.out.println(val);
                System.exit(0);
            }
        }
    }
} //:~
```

In questo esempio, grazie ad **AtomicInteger** è stato possibile eliminare la parola chiave **synchronized**. Poiché il programma non produce errori è stato aggiunto un **Timer** per interromperlo automaticamente dopo 5 secondi.

Di seguito è mostrato il codice di **MutexEvenGenerator.java** riscritto per utilizzare **AtomicInteger**.

```
//: concurrency/AtomicEvenGenerator.java
// Occasionalmente, le classi atomiche sono utili anche
// nel codice "normale".
```



```
// {RunByHand}
import java.util.concurrent.atomic.*;

public class AtomicEvenGenerator extends IntGenerator {
    private AtomicInteger currentEvenValue =
        new AtomicInteger(0);
    public int next() {
        return currentEvenValue.addAndGet(2);
    }
    public static void main(String[] args) {
        EvenChecker.test(new AtomicEvenGenerator());
    }
} // :~
```

Anche in questo caso, grazie all'impiego di **AtomicInteger** sono state eliminate tutte le altre forme di sincronizzazione.

È opportuno mettere in risalto che le classi **Atomic** sono state concepite per sviluppare le classi in **java.util.concurrent**, e che dovreste utilizzarle nel vostro codice soltanto in circostanze particolari e solo dopo esservi assicurati che non possano esservi altri problemi. Tenete presente, in ogni caso, che di solito è più sicuro fare affidamento sui lock, tramite la parola chiave **synchronized** o gli oggetti **Lock** esplicativi.

**Esercizio 14 (4)** Dimostrate che la classe **java.util.Timer** può gestire numeri elevati, creando un programma che genera numerosi oggetti **Timer** che eseguono alcuni semplici task al termine del timeout.

### **Sezioni critiche**

Talvolta potreste voler impedire che thread multipli accedano a una parte di codice all'interno di un metodo, anziché all'intero metodo. La sezione di codice da isolare in questo modo è chiamata *sezione critica*, e viene creata utilizzando la parola chiave **synchronized**. In questo caso la parola chiave **synchronized** specifica l'oggetto il cui lock è utilizzato per sincronizzare il codice incluso.

```
synchronized(syncObject) {
    // A questo codice puo' accedere
    // un solo task per volta
}
```



Questa forma sintattica è chiamata anche *blocco sincronizzato*; prima di potervi accedere, deve essere acquisito un lock su **syncObject**. Se un altro task ha già questo lock non è possibile avere accesso alla sezione critica finché il lock stesso non verrà rilasciato. Il seguente esempio confronta entrambe le tecniche di sincronizzazione, per dimostrare come il tempo a disposizione di altri task per accedere a un oggetto aumenti significativamente utilizzando un blocco **synchronized** invece della sincronizzazione dell'intero metodo. Inoltre, mostra come utilizzare una classe non protetta in un ambiente multithreaded, se controllata e protetta da un'altra classe.

```
//: concurrency/CriticalSection.java
// Sincronizzazione di blocchi invece che di interi metodi.
// Dimostra anche come proteggere una classe non thread-safe
// mediante una classe thread-safe.

package concurrency;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

class Pair { // Non sicuro a livello di thread
    private int x, y;
    public Pair(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Pair() { this(0, 0); }
    public int getX() { return x; }
    public int getY() { return y; }
    public void incrementX() { x++; }
    public void incrementY() { y++; }
    public String toString() {
        return "x: " + x + ", y: " + y;
    }
}
public class PairValuesNotEqualException
extends RuntimeException {
    public PairValuesNotEqualException() {
        super("Pair values not equal: " + Pair.this);
    }
}
```



```
// Invariante arbitraria: entrambe le variabili devono
// essere uguali:
public void checkState() {
    if(x != y)
        throw new PairValuesNotEqualException();
}

}

// Protegge un Pair all'interno di una classe non thread-safe:
abstract class PairManager {
    AtomicInteger checkCounter = new AtomicInteger(0);
    protected Pair p = new Pair();
    private List<Pair> storage =
        Collections.synchronizedList(new ArrayList<Pair>());
    public synchronized Pair getPair() {
        // Esegue una copia per conservare l'originale:
        return new Pair(p.getX(), p.getY());
    }
    // Presuppone che questa operazione richieda tempo
    protected void store(Pair p) {
        storage.add(p);
        try {
            TimeUnit.MILLISECONDS.sleep(50);
        } catch(InterruptedException ignore) {}
    }
    public abstract void increment();
}

// Sincronizza l'intero metodo:
class PairManager1 extends PairManager {
    public synchronized void increment() {
        p.incrementX();
        p.incrementY();
        store(getPair());
    }
}
```



```
// Utilizza una sezione critica:  
class PairManager2 extends PairManager {  
    public void increment() {  
        Pair temp;  
        synchronized(this) {  
            p.incrementX();  
            p.incrementY();  
            temp = getPair();  
        }  
        store(temp);  
    }  
  
    class PairManipulator implements Runnable {  
        private PairManager pm;  
        public PairManipulator(PairManager pm) {  
            this.pm = pm;  
        }  
        public void run() {  
            while(true)  
                pm.increment();  
        }  
        public String toString() {  
            return "Pair: " + pm.getPair() +  
                " checkCounter = " + pm.checkCounter.get();  
        }  
    }  
  
    class PairChecker implements Runnable {  
        private PairManager pm;  
        public PairChecker(PairManager pm) {  
            this.pm = pm;  
        }  
        public void run() {  
            while(true) {  
                pm.checkCounter.incrementAndGet();  
            }  
        }  
    }  
}
```



```
        pm.getPair().checkState();
    }
}
}

public class CriticalSection {
    // Verifica i due diversi approcci:
    static void
    testApproaches(PairManager pman1, PairManager pman2) {
        ExecutorService exec = Executors.newCachedThreadPool();
        PairManipulator
            pm1 = new PairManipulator(pman1),
            pm2 = new PairManipulator(pman2);
        PairChecker
            pcheck1 = new PairChecker(pman1),
            pcheck2 = new PairChecker(pman2);
        exec.execute(pm1);
        exec.execute(pm2);
        exec.execute(pcheck1);
        exec.execute(pcheck2);
        try {
            TimeUnit.MILLISECONDS.sleep(500);
        } catch(InterruptedException e) {
            System.out.println("Sleep interrupted");
        }
        System.out.println("pm1: " + pm1 + "\npm2: " + pm2);
        System.exit(0);
    }
    public static void main(String[] args) {
        PairManager
            pman1 = new PairManager1(),
            pman2 = new PairManager2();
        testApproaches(pman1, pman2);
    }
} /* Output: (Esempio)
pm1: Pair: x: 15, y: 15 checkCounter = 272565
pm2: Pair: x: 16, y: 16 checkCounter = 3956974
*/://:~
```



Come potete notare, **Pair** non è sicuro a livello di thread poiché la sua invarianza, evidentemente arbitraria, richiede che entrambe le variabili abbiano lo stesso valore. Inoltre, come si è visto in precedenza in questo capitolo, anche le operazioni di incremento non sono sicure a livello di thread, e poiché nessuno dei metodi è **synchronized** non è possibile confidare che un oggetto **Pair** rimanga inalterato (*uncorrupted*) in un programma a thread.

Immaginate che qualcuno vi passi la classe **Pair** non sicura e che dobbiate utilizzarla in un ambiente a thread. Per fare questo create la classe **PairManager**, che contiene un oggetto **Pair** e ne gestisce l'accesso. Notate che gli unici metodi **public** sono **getPair()**, che è **synchronized**, e **increment()**, definito come **abstract**. La sincronizzazione per **increment()** sarà gestita quando verrà implementata.

La struttura di **PairManager**, la cui funzionalità implementata nella classe di base utilizza uno o più metodi **abstract** che sono definiti nelle classi derivate, in gergo viene chiamata *Template Method* (*Design Patterns*, di Gamma, Helm, Johnson e Vlissides, Addison-Wesley, 1995). I design pattern consentono di incapsulare le modifiche nel codice; in questo caso, la parte che viene cambiata è il metodo **increment()**. In **PairManager1** l'intero metodo **increment()** viene sincronizzato (**synchronized**), mentre in **PairManager2** solo una parte di **increment()** è sincronizzata tramite un blocco **synchronized**. Notate che la parola chiave **synchronized** non fa parte della segnatura del metodo, di conseguenza può essere aggiunta durante la sovrascrittura.

Il metodo **store()** aggiunge un oggetto **Pair** a **synchronized ArrayList**, così che questa operazione sia sicura a livello di thread: pertanto il metodo non deve essere controllato e viene posto all'esterno del blocco **synchronized** presente in **PairManager2**.

**PairManipulator** viene creato per testare i due diversi tipi di **PairManager**, chiamando il metodo **increment()** in un task mentre **PairChecker** viene eseguito da un altro task. Per tenere traccia del numero di test riusciti, **PairChecker** incrementa **checkCounter** ogni volta che l'operazione ha successo. In **main()** vengono creati due oggetti **PairManipulator**, poi eseguiti per un certo tempo; in seguito i risultati ottenuti da ogni **PairManipulator** sono visualizzati.

Anche se probabilmente noterete molte variazioni nell'output di ogni esecuzione, in genere vedrete che **PairManager1.increment()** non consente di accedere a **PairChecker** quanto **PairManager2.increment()**; infatti, quest'ultimo metodo possiede il blocco **synchronized** e per questo garantisce un tempo "unlocked" maggiore. Questo è il tipico motivo per utilizzare un blocco **synchronized** invece di sincronizzare l'intero metodo: concedere ad altri task maggiori possibilità di accesso (nei limiti imposti dai vincoli di



sicurezza). Per creare sezioni critiche, potete anche servirvi di oggetti **Lock** esplicativi.

```
//: concurrency/ExplicitCriticalSection.java
// Utilizzo di oggetti Lock esplicativi per creare sezioni
// critiche.
package concurrency;
import java.util.concurrent.locks.*;

// Sincronizza l'intero metodo:
class ExplicitPairManager1 extends PairManager {
    private Lock lock = new ReentrantLock();
    public synchronized void increment() {
        lock.lock();
        try {
            p.incrementX();
            p.incrementY();
            store(getPair());
        } finally {
            lock.unlock();
        }
    }
}

// Utilizza una sezione critica:
class ExplicitPairManager2 extends PairManager {
    private Lock lock = new ReentrantLock();
    public void increment() {
        Pair temp;
        lock.lock();
        try {
            p.incrementX();
            p.incrementY();
            temp = getPair();
        } finally {
            lock.unlock();
        }
    }
}
```



```
        store(temp);
    }
}

public class ExplicitCriticalSection {
    public static void main(String[] args) throws Exception {
        PairManager
            pman1 = new ExplicitPairManager1(),
            pman2 = new ExplicitPairManager2();
        CriticalSection.testApproaches(pman1, pman2);
    }
} /* Output: (Esempio)
pman1: Pair: x: 15, y: 15 checkCounter = 174035
pman2: Pair: x: 16, y: 16 checkCounter = 2608588
*///:-
```

Questo codice riutilizza la maggior parte di **CriticalSection.java** e crea i nuovi tipi **PairManager**, che utilizzano gli oggetti **Lock** esplicativi. In **ExplicitPairManager2** potete vedere la creazione di una sezione critica mediante un oggetto **Lock**; la chiamata al metodo **store()** è esterna alla sezione critica.

### *Sincronizzazione su altri oggetti*

A un blocco **synchronized** deve essere fornito un oggetto con il quale sincronizzarsi; il più adatto solitamente è l'oggetto corrente per cui il metodo viene chiamato, **synchronized(this)**, che è poi la tecnica adottata in **PairManager2**. In questo modo, quando viene acquisito il lock per il blocco **synchronized**, altri metodi **synchronized** e sezioni critiche nell'oggetto non possono essere chiamati. Pertanto, l'effetto di una sezione critica che si sincronizza su **this** è semplicemente la riduzione dell'ambito di sincronizzazione.

A volte potreste avere la necessità di sincronizzarvi su un altro oggetto, ma se lo fate dovrete accertarvi che tutti i relativi task stiano sincronizzandosi sullo stesso oggetto. L'esempio seguente dimostra che due task possono accedere a un oggetto quando i metodi di tale oggetto si sincronizzano su lock differenti.

```
//: concurrency/SyncObject.java
// Sincronizzazione su un altro oggetto.
import static net.mindview.util.Print.*;
```



```
class DualSynch {  
    private Object syncObject = new Object();  
    public synchronized void f() {  
        for(int i = 0; i < 5; i++) {  
            print("f()");  
            Thread.yield();  
        }  
    }  
    public void g() {  
        synchronized(syncObject) {  
            for(int i = 0; i < 5; i++) {  
                print("g()");  
                Thread.yield();  
            }  
        }  
    }  
}  
  
public class SyncObject {  
    public static void main(String[] args) {  
        final DualSynch ds = new DualSynch();  
        new Thread() {  
            public void run() {  
                ds.f();  
            }  
        }.start();  
        ds.g();  
    }  
} /* Output: (Esempio)  
g()  
f()  
g()  
f()  
g()  
f()  
g()  
f()
```



```
g()
f()
*///:~
```

**DualSync.f()** sincronizza **this**, sincronizzando l'intero metodo, mentre **g()** ha un blocco **synchronized** che si sincronizza su **syncObject**: le due sincronizzazioni sono quindi indipendenti. Questo viene dimostrato in **main()**, creando un **Thread** che chiama **f()**; il thread **main()** è utilizzato per chiamare **g()**. L'output mostra che entrambi i metodi operano simultaneamente e che nessuno è bloccato dalla sincronizzazione dell'altro.

**Esercizio 15** (1) Create una classe con tre metodi contenenti sezioni critiche sincronizzate tutte sullo stesso oggetto. Generate poi task multipli per dimostrare che può essere eseguito soltanto un metodo per volta. Infine modificate i metodi in modo che ciascuno si sincronizzi su un oggetto differente, e dimostrate che i tre i metodi possono essere eseguiti contemporaneamente.

**Esercizio 16** (1) Modificate l'Esercizio 15 per utilizzare oggetti **Lock** esplicativi.

### **Memoria locale di thread**

Una seconda tecnica per impedire ai task di collidere a fronte di risorse condivise consiste nell'eliminare la condivisione delle variabili. La memoria locale di thread (*Thread local storage*) è un meccanismo che crea automaticamente diverse aree di memorizzazione per la stessa variabile, per ogni thread che impiega un oggetto. Quindi, se avete cinque thread che utilizzano un oggetto con una variabile **x**, la memoria locale di thread genera cinque aree di memoria differenti per questa variabile, consentendovi, di fatto, di associare lo stato a un thread.

La creazione e la gestione della memoria locale di thread sono eseguite dalla classe **java.lang.ThreadLocal**, come vedete nell'esempio seguente.

```
//: concurrency/ThreadLocalVariableHolder.java
// Fornisce automaticamente a ciascun thread la sua area
// di memorizzazione.
import java.util.concurrent.*;
import java.util.*;

class Accessor implements Runnable {
    private final int id;
```



```
public Accessor(int idn) { id = idn; }
public void run() {
    while(!Thread.currentThread().isInterrupted()) {
        ThreadLocalVariableHolder.increment();
        System.out.println(this);
        Thread.yield();
    }
}
public String toString() {
    return "#" + id + ":" +
        ThreadLocalVariableHolder.get();
}
}

public class ThreadLocalVariableHolder {
    private static ThreadLocal<Integer> value =
        new ThreadLocal<Integer>() {
        private Random rand = new Random(47);
        protected synchronized Integer initialValue() {
            return rand.nextInt(10000);
        }
    };
    public static void increment() {
        value.set(value.get() + 1);
    }
    public static int get() { return value.get(); }
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new Accessor(i));
        TimeUnit.SECONDS.sleep(3); // Disattiva per il tempo
                                // indicato
        exec.shutdownNow();      // Tutti gli oggetti Accessors
                                // vengono chiusi
    }
} /* Output: (Esempio)
#0: 9259
```



```
#1: 556  
#2: 6694  
#3: 1862  
#4: 962  
#0: 9260  
#1: 557  
#2: 6695  
#3: 1863  
#4: 963  
...  
*///:~
```

Normalmente gli oggetti **ThreadLocal** vengono memorizzati come campi **static**. Quando create un oggetto **ThreadLocal** potete accedere al suo contenuto soltanto utilizzando i metodi **get()** e **set()**. Il metodo **get()** restituisce una copia dell'oggetto associato a quel thread, mentre **set()** inserisce il relativo argomento nell'oggetto memorizzato per quel thread, restituendo il vecchio oggetto che era presente in memoria.

I metodi **increment()** e **get()** dimostrano questo meccanismo nella classe **ThreadLocalVariableHolder**. Notate che **increment()** e **get()** non sono **synchronized**, poiché **ThreadLocal** garantisce che la “corsa critica” non abbia luogo.

Eseguendo questo programma risulta evidente che i diversi thread sono allocati ciascuno in un'area di memoria autonoma, poiché ogni thread mantiene il suo conteggio anche se esiste un solo oggetto **ThreadLocalVariableHolder**.

## Chiusura dei task

In alcuni esempi precedenti, i metodi **cancel()** e **isCancelled()** sono stati inseriti in una classe che era visibile da tutti i task. I task controllano **isCancelled()** per determinare quando terminare. Questo è un approccio ragionevole, ma in alcuni casi è necessario che un task termini bruscamente. Per prima cosa occorre esaminare un esempio che non solo illustra il problema della terminazione, ma costituisce un'ulteriore dimostrazione della condivisione di risorse.

### *Il giardino ornamentale*

In questa simulazione, si ipotizza che la direzione di un “giardino ornamentale” voglia sapere quante persone entrano ogni giorno transitando dai vari cancelli. Ogni entrata ha un tornello o un altro tipo di dispositivo conta-ac-



cessi; dopo che il conteggio del tornello è stato incrementato, lo è anche un conteggio condiviso che rappresenta il numero totale di visitatori.

```
//: concurrency/OrnamentalGarden.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Count {
    private int count = 0;
    private Random rand = new Random(47);
    // Rimuovere la parola chiave synchronized per vedere come
    // il conteggio fallisce:
    public synchronized int increment() {
        int temp = count;
        if(rand.nextBoolean()) // Esegue yield per metà del tempo
            Thread.yield();
        return (count = ++temp);
    }
    public synchronized int value() { return count; }
}

class Entrance implements Runnable {
    private static Count count = new Count();
    private static List<Entrance> entrances =
        new ArrayList<Entrance>();
    private int number = 0;
    // Non necessita di sincronizzazione per la lettura:
    private final int id;
    private static volatile boolean canceled = false;
    // Operazione atomica su un campo volatile:
    public static void cancel() { canceled = true; }
    public Entrance(int id) {
        this.id = id;
        // Mantiene questo task in un elenco. Evita anche
        // la garbage collection dei task "deceduti":
        entrances.add(this);
    }
}
```



```
public void run() {
    while(!canceled) {
        synchronized(this) {
            ++number;
        }
        print(this + " Total: " + count.increment());
        try {
            TimeUnit.MILLISECONDS.sleep(100);
        } catch(InterruptedException e) {
            print("sleep() interrupted");
        }
    }
    print("Stopping " + this);
}
public synchronized int getValue() { return number; }
public String toString() {
    return "Entrance " + id + ":" + getValue();
}
public static int getTotalCount() {
    return count.value();
}
public static int sumEntrances() {
    int sum = 0;
    for(Entrance entrance : entrances)
        sum += entrance.getValue();
    return sum;
}
}

public class OrnamentalGarden {
public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
        exec.execute(new Entrance(i));
    // Funziona per un certo tempo, poi si ferma e raccoglie
    // i dati:
    TimeUnit.SECONDS.sleep(3);
}
```



```
    Entrance.cancel();
    exec.shutdown();
    if(!exec.awaitTermination(250, TimeUnit.MILLISECONDS))
        print("Some tasks were not terminated!");
    print("Total: " + Entrance.getTotalCount());
    print("Sum of Entrances: " + Entrance.sumEntrances());
}
} /* Output: (Esempio)
Entrance 0: 1 Total: 1
Entrance 2: 1 Total: 3
Entrance 1: 1 Total: 2
Entrance 4: 1 Total: 5
Entrance 3: 1 Total: 4
Entrance 2: 2 Total: 6
Entrance 4: 2 Total: 7
Entrance 0: 2 Total: 8
...
Entrance 3: 29 Total: 143
Entrance 0: 29 Total: 144
Entrance 4: 29 Total: 145
Entrance 2: 30 Total: 147
Entrance 1: 30 Total: 146
Entrance 0: 30 Total: 149
Entrance 3: 30 Total: 148
Entrance 4: 30 Total: 150
Stopping Entrance 2: 30
Stopping Entrance 1: 30
Stopping Entrance 0: 30
Stopping Entrance 3: 30
Stopping Entrance 4: 30
Total: 150
Sum of Entrances: 150
*///:~
```

Un singolo oggetto **Count** mantiene il conteggio globale dei visitatori del giardino, ed è memorizzato come campo **static** nella classe **Entrance**. I metodi **Count.increment()** e **Count.value()** sono **synchronized** per controllare



l'accesso al campo **count**. Il metodo **increment()** utilizza un oggetto **Random** per generare una "precedenza" (**yield()**) all'incirca metà delle volte, tra la copia del valore di **count** in **temp** e l'incremento e la nuova archiviazione di **temp** in **count**. Se commentate la parola chiave **synchronized** in **increment()**, il programma si interromperà poiché diversi task cercheranno di accedere a **count** e di modificarlo: **yield()** fa sì che il problema accada più rapidamente.

Ogni task **Entrance** mantiene un valore locale **number** con il numero di visitatori che hanno utilizzato un determinato cancello. Questo fornisce una doppia verifica con l'oggetto **count** per assicurarsi che si stia registrando l'esatto numero di visitatori. **Entrance.run()** incrementa **number** e l'oggetto **count**, poi mette il task in pausa per 100 millisecondi.

Poiché **Entrance.canceled** è un flag **volatile boolean** che viene soltanto letto e assegnato, senza essere mai letto insieme ad altri campi, è possibile evitare di sincronizzarne l'accesso: in caso di dubbio, tuttavia, è sempre opportuno utilizzare **synchronized**.

Questo programma richiede qualche accorgimento per chiudere tutto in modo "pulito". In parte questo è dovuto alla necessità di dimostrarvi quanta attenzione occorra dedicare alla chiusura di un programma multithreaded, in parte per mostrarvi il valore del metodo **interrupt()**, di cui si parlerà tra breve. Dopo 3 secondi di esecuzione, **main()** invia il messaggio **static cancel()** a **Entrance**, quindi chiama **shutdown()** per l'oggetto **exec**, infine chiama **awaitTermination()** su **exec**. Il metodo **ExecutorService.awaitTermination()** attende che ogni task sia completo: se tutto viene completato prima dello scadere del valore di **timeout**, restituisce **true**, altrimenti restituisce **false** per indicare che alcuni task non sono stati completati. Anche se questo fa in modo che ogni task esca dal suo metodo **run()** e che quindi termini come task, gli oggetti **Entrance** sono ancora validi poiché, nel costruttore, ogni oggetto di **Entrance** è memorizzato in un oggetto **static List<Entrance>** chiamato **entrances**. Pertanto, **sumEntrances()** sta ancora lavorando con oggetti **Entrance** validi.

Quando eseguirete questo programma vi saranno visualizzati il conteggio totale e il conteggio di ogni entrata, corrispondenti al numero di visitatori che attraversano un tornello. Se rimuovete la dichiarazione **synchronized** per **Count.increment()**, noterete che il numero totale di visitatori non è quello che vi aspettereste. Il numero di accessi contati da ogni tornello sarà diverso dal valore presente in **count**. Finché il mutex sincronizza l'accesso a **Count**, tutto funziona correttamente. Tenete presente che **Count.increment()** esaspera il potenziale problema utilizzando **temp** e **yield()**. Nei problemi di threading reali la probabilità di errore potrebbe essere statisticamente inferiore, ed è



quindi facile cadere nella trappola di ritenere che il programma stia funzionando correttamente. Come nell'esempio precedente, possono esservi problemi che non si sono ancora presentati, quindi dovete essere estremamente diligenti quando ricontrollate il codice concorrente.

**Esercizio 17 (2)** Create un contatore Geiger che possa avere un numero arbitrario di sensori remoti.

### Terminare un task bloccato

Il metodo **Entrance.run()** nell'esempio precedente include nel ciclo una chiamata a **sleep()**. Sapete che **sleep()** termina dopo un certo tempo, pertanto il task raggiungerà la parte superiore del ciclo, dove avrà occasione di terminarlo controllando il flag **canceled**. Tuttavia, **sleep()** è soltanto una delle situazioni in cui l'esecuzione di un task è bloccata; talvolta possono esservene altre che richiedono di terminare un task bloccato.

### Stati dei thread

Generalmente, un thread si trova in una delle seguenti condizioni.

1. *New* (nuovo): un thread rimane in questa condizione solo temporaneamente, durante la sua creazione. Esso alloca ogni risorsa di sistema necessaria, ed esegue l'inizializzazione; a quel punto può essere preso in carico dalla CPU. Il pianificatore gestisce la transizione di questo thread negli stati *runnable* o *blocked*.
2. *Runnable* (eseguibile): significa che un thread può essere eseguito quando il meccanismo di suddivisione dei tempi (*time slicing*) ha cicli CPU disponibili. In questo stato, quindi, il thread è nella condizione di essere eseguito in qualsiasi momento, e niente gli impedisce di essere eseguito se il pianificatore può gestirlo: in pratica, non è né *dead* né *blocked*.
3. *Blocked* (bloccato): il thread può essere potenzialmente eseguito, ma qualcosa lo impedisce. Mentre un thread si trova in questo stato è ignorato dal pianificatore, che non gli assegna cicli di CPU. Finché il thread non ritorna allo stato runnable non eseguirà nessuna operazione.
4. *Dead* o *Terminated* (deceduto): un thread in questo stato non è più pianificabile e non riceverà nessun ciclo di CPU. Il suo task è completato e non è più eseguibile. Un task può passare nello stato dead ritornando dal metodo **run()**, ma il thread del task può anche essere interrotto, come vedrete tra breve.<sup>11</sup>

11. Per l'elenco completo degli stati dei thread aggiornati a Java SE5, consultate la documentazione di Sun all'indirizzo <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.State.html>.



## Condizione di blocco

Un task può entrare nello stato blocked per i motivi elencati di seguito.

1. Il task è stato messo in pausa chiamando **sleep(milliseconds)**, nel qual caso non verrà eseguito prima dello scadere del tempo specificato.
2. È stata sospesa con **wait()** l'esecuzione del thread, che non diventerà nuovamente runnable finché non otterrà il messaggio **notify()** o **notifyAll()**, o gli equivalenti **signal()** e **signalAll()** per gli strumenti della libreria Java SE5 **java.util.concurrent**. Questi ultimi saranno esaminati nel prosieguo del capitolo.
3. Il task sta attendendo il completamento delle operazioni di I/O.
4. Il task sta tentando di chiamare un metodo **synchronized** su un altro oggetto, e il lock su tale oggetto non è disponibile poiché è stato già acquisito da un altro task.

Nel vecchio codice potreste scoprire che per bloccare e sbloccare i thread sono stati utilizzati i metodi **suspend()** e **resume()**: questi metodi, il cui utilizzo è sconsigliato nelle implementazioni di Java più moderne poiché inclini a bloccarsi definitivamente (condizione di *deadlock*), non verranno esaminati in questo manuale. Anche il metodo **stop()** è deprecato, poiché non rilascia i lock acquisiti dal thread: se gli oggetti sono in uno stato inconsistente (*damaged*), altri task potrebbero accedervi e modificarli anche in questo stato, provocando problemi difficili da rilevare.

In questo momento il problema da valutare è il seguente: potreste voler terminare un task che si trova in stato blocked. Se non potete aspettare che il task arrivi a un punto del codice che permetta di controllarne lo stato e fare in modo che termini naturalmente, dovrete forzare il task a uscire dallo stato blocked.

## Interruzione

Come potete immaginare, è molto più caotico interrompere all'improvviso un metodo **Runnable.run()** che attendere che questo metodo arrivi a valutare il proprio flag "cancel", o che giunga in qualche altro punto in cui il programma è pronto per uscire dal metodo. Quando interrompere un task bloccato potreste avere bisogno di eseguire il cleanup delle risorse; per questo motivo l'interruzione del metodo **run()** di un task è più simile all'azione di sollevare un'eccezione che a qualsiasi altro meccanismo. Nei thread quindi, per questo tipo di chiusura dei task, Java ricorre alle eccezioni.<sup>12</sup>

12. La comunicazione delle eccezioni non avviene mai in modo asincrono, pertanto non vi è pericolo che qualcosa faccia interrompere chiamate di metodo a metà istruzione. Inoltre, se applicate il blocco **try-finally** quando utilizzate i mutex dell'oggetto (invece della parola chiave **synchronized**), essi verranno automaticamente rilasciati in caso di eccezione.



Tenete presente che questa tecnica, tuttavia, oltrepassa il sottile confine dell'utilizzo inadeguato delle eccezioni, in quanto se ne serve per controllare il flusso operativo. Quando terminate un task in questo modo, per tornare a una condizione operativa nota dovete considerare con attenzione i percorsi di esecuzione del codice e scrivere le clausole **catch** opportune che eseguano le operazioni di cleanup corrette.

Per consentire di terminare un task bloccato la classe **Thread** offre il metodo **interrupt()**, che imposta lo stato di interruzione (*interrupted*) per il thread in questione. Un thread nella condizione di interrupted solleva una **InterruptedException** se è già bloccato o se cerca di eseguire un'operazione bloccante. Lo stato interrupted verrà azzerato nel momento in cui sarà gestita l'eccezione o se il task chiama **Thread.interrupted()**. Come vedrete, l'utilizzo di **Thread.interrupted()** costituisce un'altra tecnica per uscire dal ciclo **run()** senza sollevare un'eccezione. Per chiamare **interrupt()** dovete possedere un oggetto **Thread**. Forse avete notato che la nuova libreria **concurrent** sembra evitare la manipolazione diretta degli oggetti **Thread**, privilegiando le operazioni tramite gli **Executor**. Se chiamate il metodo **shutdownNow()** su un **Executor**, esso invierà una chiamata **interrupt()** a ciascuno dei thread che ha avviato.

Questo comportamento ha senso poiché di norma interromperete tutti i task di un determinato **Executor** immediatamente, non appena avrete terminato parte di un progetto o l'intero programma. Tuttavia, in alcune situazioni potreste voler interrompere una sola operazione. Se state utilizzando oggetti **Executor** potete mantenere il contesto di un task quando lo avviate, chiamando il metodo **submit()** anziché **execute()**. Il metodo **submit()** restituisce un generico **Future<?>**, con un parametro non specificato poiché non chiamerete mai **get()**; il motivo per conservare questo tipo di **Future** è che potete chiamare il metodo **cancel()** e utilizzarlo per interrompere un particolare task. Se passate **true** a **cancel()** consentite la chiamata **interrupt()** su quel thread per interromperlo; così, il metodo **cancel()** è un modo per interrompere i diversi thread avviati per mezzo di un **Executor**. L'esempio seguente mostra i principi fondamentali di **interrupt()** utilizzando gli **Executor**.

```
//: concurrency/Interrupting.java
// Come interrompere un thread bloccato.
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;
```

```
class SleepBlocked implements Runnable {
    public void run() {
```



```
try {
    TimeUnit.SECONDS.sleep(100);
} catch(InterruptedException e) {
    print("InterruptedException");
}
print("Exiting SleepBlocked.run()");
}

class IOBlocked implements Runnable {
    private InputStream in;
    public IOBlocked(InputStream is) { in = is; }
    public void run() {
        try {
            print("Waiting for read():");
            in.read();
        } catch(IOException e) {
            if(Thread.currentThread().isInterrupted()) {
                print("Interrupted from blocked I/O");
            } else {
                throw new RuntimeException(e);
            }
        }
        print("Exiting IOBlocked.run()");
    }
}

class SynchronizedBlocked implements Runnable {
    public synchronized void f() {
        while(true) // Non rilascia mai il lock
            Thread.yield();
    }
    public SynchronizedBlocked() {
        new Thread() {
            public void run() {
                f(); // Lock acquisito dal thread corrente
            }
        }
    }
}
```



```
    }.start();
}
public void run() {
    print("Trying to call f()");
    f();
    print("Exiting SynchronizedBlocked.run()");
}
}

public class Interrupting {
    private static ExecutorService exec =
        Executors.newCachedThreadPool();
    static void test(Runnable r) throws InterruptedException{
        Future<?> f = exec.submit(r);
        TimeUnit.MILLISECONDS.sleep(100);
        print("Interrupting " + r.getClass().getName());
        f.cancel(true); // Interrompe se in esecuzione
        print("Interrupt sent to " + r.getClass().getName());
    }
    public static void main(String[] args) throws Exception {
        test(new SleepBlocked());
        test(new IOBlocked(System.in));
        test(new SynchronizedBlocked());
        TimeUnit.SECONDS.sleep(3);
        print("Abortinf with System.exit(0)");
        System.exit(0); // ... perche' gli ultimi 2 interrupt
                        // sono falliti
    }
} /* Output: (95% match)
Interrupting SleepBlocked
InterruptedException
Exiting SleepBlocked.run()
Interrupt sent to SleepBlocked
Waiting for read():
Interrupting IOBlocked
Interrupt sent to IOBlocked
Trying to call f()
```



```
Interrupting SynchronizedBlocked  
Interrupt sent to SynchronizedBlocked  
Aborting with System.exit(0)  
*///:-
```

Ogni task rappresenta un tipo differente di bloccaggio. **SleepBlock** è un esempio di bloccaggio che può essere interrotto, mentre **IOBlocked** e **SynchronizedBlocked** non sono interrompibili.<sup>13</sup>

Il programma dimostra che l'attività di I/O e l'attesa su un lock **synchronized** non sono interrompibili, ma potete anche intuirlo osservando il codice: non è richiesto alcun gestore di **InterruptedException**, né per l'I/O né per cercare di chiamare un metodo **synchronized**.

Le prime due classi sono semplici: il metodo **run()** chiama **sleep()** nella prima classe e **read()** nella seconda. Per dimostrare **SynchronizedBlocked**, tuttavia, dovete in primo luogo acquisire il lock direttamente nel costruttore, creando un'istanza di una classe anonima **Thread** che acquisisce il lock dell'oggetto chiamando il metodo **f()**. Tenete presente che il thread deve essere diverso da quello che pilota **run()** per **SynchronizedBlock**, dal momento che un thread può acquisire più volte il lock su un oggetto. Poiché **f()** non ritorna, il lock non viene mai rilasciato. **SynchronizedBlock.run()** tenta di chiamare **f()** e rimane bloccato in attesa che il lock venga rilasciato.

Noterete dall'output che è possibile interrompere una chiamata **sleep()** o qualsiasi altra che richieda di sollevare una **InterruptedException**; non potete però interrompere un task che sta tentando di acquisire un lock **synchronized** né un task che cerchi di eseguire operazioni di I/O. Questa regola è alquanto sconcertante, specialmente se state creando un task che esegue operazioni di I/O, poiché significa che l'I/O può bloccare programmi multithreaded: una fonte di problemi, soprattutto per le applicazioni web.

Una soluzione complessa ma talvolta efficace a questo problema consiste nel chiudere la risorsa sottostante sulla quale si è bloccato il task.

```
//: concurrency/CloseResource.java  
// Interruzione di un task bloccato mediante chiusura della  
// risorsa sottostante.  
// {RunByHand}
```

13. Alcune versioni del JDK offrono anche il supporto per **InterruptedException**, che tuttavia è un'implementazione parziale e valida solo per alcune piattaforme. Se questa eccezione viene sollevata farà sì che oggetti di IO divengano inutilizzabili. È improbabile che le versioni future di JDK continuino a supportare questa eccezione.



```
import java.net.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class CloseResource {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        ServerSocket server = new ServerSocket(8080);
        InputStream socketInput =
            new Socket("localhost", 8080).getInputStream();
        exec.execute(new IOBlocked(socketInput));
        exec.execute(new IOBlocked(System.in));
        TimeUnit.MILLISECONDS.sleep(100);
        print("Shutting down all threads");
        exec.shutdownNow();
        TimeUnit.SECONDS.sleep(1);
        print("Closing " + socketInput.getClass().getName());
        socketInput.close(); // Rilascia il thread bloccato
        TimeUnit.SECONDS.sleep(1);
        print("Closing " + System.in.getClass().getName());
        System.in.close(); // Rilascia il thread bloccato
    }
} /* Output: (85% match)
Waiting for read():
Waiting for read():
Shutting down all threads
Closing java.net.SocketInputStream
Interrupted from blocked I/O
Exiting IOBlocked.run()
Closing java.io.BufferedReader
Exiting IOBlocked.run()
*///:~
```

Dopo la chiamata a `shutdownNow()`, i ritardi impostati prima di chiamare `close()` sui due flussi di input evidenziano il fatto che i task si sbloccano quando la risorsa sottostante viene chiusa. È interessante notare che `interrupt()` compare quando state terminando un `Socket` ma non quando terminate `System.in`.



Fortunatamente le classi **nio** (definite nel Volume 2, Capitolo 6) consentono un'interruzione più “garbata” di queste operazioni. I canali **nio** bloccati rispondono automaticamente alle interruzioni.

```
//: concurrency/NIO INTERRUPTION.java
// Interruzione di un canale NIO bloccato.
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class NIOBlocked implements Runnable {
    private final SocketChannel sc;
    public NIOBlocked(SocketChannel sc) { this.sc = sc; }
    public void run() {
        try {
            print("Waiting for read() in " + this);
            sc.read(ByteBuffer.allocate(1));
        } catch(ClosedByInterruptException e) {
            print("ClosedByInterruptException");
        } catch(AsynchronousCloseException e) {
            print("AsynchronousCloseException");
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        print("Exiting NIOBlocked.run() " + this);
    }
}

public class NIO INTERRUPTION {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        ServerSocket server = new ServerSocket(8080);
        InetSocketAddress isa =
            new InetSocketAddress("localhost", 8080);
    }
}
```



```

        SocketChannel sc1 = SocketChannel.open(isa);
        SocketChannel sc2 = SocketChannel.open(isa);
        Future<?> f = exec.submit(new NIOBlocked(sc1));
        exec.execute(new NIOBlocked(sc2));
        exec.shutdown();
        TimeUnit.SECONDS.sleep(1);
        // Genera un interrupt tramite cancel:
        f.cancel(true);
        TimeUnit.SECONDS.sleep(1);
        // Rilascia il blocco chiudendo il canale:
        sc2.close();
    }
} /* Output: (Esempio)
Waiting for read() in NIOBlocked@7a84e4
Waiting for read() in NIOBlocked@15c7850
ClosedByInterruptException
Exiting NIOBlocked.run() NIOBlocked@15c7850
AsynchronousCloseException
Exiting NIOBlocked.run() NIOBlocked@7a84e4
*///:~

```

Come vedete, potete anche chiudere il canale sottostante per rilasciare il blocco, benché questa possibilità risulti raramente utile. Notate che utilizzando **execute()** per avviare entrambi i task e chiamando **e.shutdownNow()** chiudrete senza problemi; nell'esempio precedente, l'intercettazione di **Future** è necessaria solo per trasmettere l'interrupt a un thread, non all'altro.<sup>14</sup>

**Esercizio 18 (2)** Create una classe non task contenente un metodo che chiama **sleep()** per un lungo intervallo di tempo. Generate un task che chiama il metodo nella classe non task. Nel metodo **main()** avviate il task, quindi chiamate **interrupt()** per terminarlo. Assicuratevi che il task si interrompa in modo sicuro.

**Esercizio 19 (4)** Modificate **OrnamentalGarden.java** in modo che utilizzi il metodo **interrupt()**.

**Esercizio 20 (1)** Modificate **CachedThreadPool.java** in modo che tutti i task ricevano un **interrupt()** prima di essere completati.

---

14. Ervin Varga ha collaborato alle ricerche per la stesura di questi paragrafi.



### Blocco causato da mutex

Come avete visto in [Interrupting.java](#), se provate a chiamare un metodo **synchronized** su un oggetto il cui lock sia già stato acquisito, il task chiamante rimarrà sospeso (bloccato) finché il lock non diventerà disponibile. Questo esempio mostra come lo stesso mutex può essere acquisito più volte dallo stesso task.

```
//: concurrency/MultiLock.java
// Un thread puo' acquisire piu' volte lo stesso lock.
import static net.mindview.util.Print.*;

public class MultiLock {
    public synchronized void f1(int count) {
        if(count-- > 0) {
            print("f1() calling f2() with count " + count);
            f2(count);
        }
    }
    public synchronized void f2(int count) {
        if(count-- > 0) {
            print("f2() calling f1() with count " + count);
            f1(count);
        }
    }
    public static void main(String[] args) throws Exception {
        final MultiLock multiLock = new MultiLock();
        new Thread() {
            public void run() {
                multiLock.f1(10);
            }
        }.start();
    }
} /* Output:
f1() calling f2() with count 9
f2() calling f1() with count 8
f1() calling f2() with count 7
f2() calling f1() with count 6
f1() calling f2() with count 5
```



```
f2() calling f1() with count 4
f1() calling f2() with count 3
f2() calling f1() with count 2
f1() calling f2() with count 1
f2() calling f1() with count 0
*///:~
```

In **main()** viene creato un **Thread** per chiamare **f1()**, quindi **f1()** e **f2()** si chiamano a vicenda fino a quando **count** non arriva a zero. Dal momento che l'operazione ha già acquisito il lock dell'oggetto **multiLock** all'interno della prima chiamata a **f1()**, lo stesso task lo acquisisce nuovamente nella chiamata a **f2()** e così via. Questo ha senso poiché un task deve essere in grado di chiamare altri metodi **synchronized** all'interno dello stesso oggetto; quel task possiede già il lock.

Come osservato in precedenza nel caso dell'I/O non interrompibile, quando un task può bloccarsi in modo da non poter essere interrotto, avete la possibilità di "congelare" un programma. Una delle novità delle librerie di concorrenza in Java SE5 è la capacità di interrompere i task bloccati su **ReentrantLock**, diversamente dai task bloccati sui metodi o sulle sezioni critiche **synchronized**.

```
//: concurrency/Interrupting2.java
// Interruzione di un task bloccato, con un ReentrantLock.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class BlockedMutex {
    private Lock lock = new ReentrantLock();
    public BlockedMutex() {
        // Acquisisce subito il lock, per dimostrare
        // l'interruzione di un task bloccato su un ReentrantLock:
        lock.lock();
    }
    public void f0() {
        try {
            // Questo non sara' mai disponibile a un secondo task
            lock.lockInterruptibly(); // Chiamata speciale
            print("lock acquired in f0()");
        } catch(InterruptedException e) {
```



```
        print("Interrupted from lock acquisition in f()");
    }
}
}

class Blocked2 implements Runnable {
    BlockedMutex blocked = new BlockedMutex();
    public void run() {
        print("Waiting for f() in BlockedMutex");
        blocked.f();
        print("Broken out of blocked call");
    }
}

public class Interrupting2 {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new Blocked2());
        t.start();
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Issuing di t.interrupt()");
        t.interrupt();
    }
} /* Output:
Waiting for f() in BlockedMutex
Issuing t.interrupt()
Interrupted from lock acquisition in f()
Broken out of blocked call
*///:~
```

La classe **BlockedMutex** ha un costruttore che acquisisce il **Lock** dell'oggetto, senza mai rilasciarlo. Per questo motivo, se provate a chiamare **f()** da un secondo task (diverso da quello che ha creato **BlockedMutex**) sarete sempre bloccati poiché **Mutex** non può essere acquisito. In **Blocked2** il metodo **run()** sarà interrotto durante la chiamata a **blocked.f()**. Quando eseguirete il programma osserverete che, a differenza di quanto avviene con una chiamata di I/O, **interrupt()** può uscire da una chiamata bloccata da un mutex.<sup>15</sup>

15. Tenete presente che, per quanto sia improbabile, la chiamata a **t.interrupt()** potrebbe effettivamente verificarsi prima della chiamata a **blocked.f()**.



## Controllo di un interrupt

Notate che quando chiamate **interrupt()** su un thread l'unica situazione in cui si verifica l'interruzione è quando l'operazione accede a un'operazione bloccante, o vi si trova già: questo, tranne il caso di I/O non interrompibile o di metodi **synchronized** bloccati, poiché in tale eventualità non potrete fare nulla. Ma che cosa accade se scrivete codice che potrebbe o non potrebbe eseguire una chiamata bloccante, a seconda delle condizioni in cui viene eseguita? Se potete uscire soltanto sollevando un'eccezione su una chiamata bloccante, non sarete sempre in grado di lasciare il ciclo **run()**. Quindi, se chiamate il metodo **interrupt()** per terminare un task, il vostro task dovrà avere una seconda possibilità di uscita, qualora il vostro ciclo **run()** non esegua alcuna chiamata bloccante.

Questa opportunità è rappresentata dallo *stato di interrupt (interrupted status)*, che è impostato dalla chiamata a **interrupt()** e può essere verificato chiamando il metodo **interrupted()**. In questo modo non soltanto saprete se **interrupt()** è stato chiamato, ma potrete anche annullare lo stato di interrupt. L'annullamento di questo stato assicura che il framework non vi informi due volte di un'operazione che è stata interrotta: sarete informati tramite una sola **InterruptedException** o mediante un solo test **Thread.interrupted()** riuscito. Qualora vogliate verificare nuovamente se si è verificata un'interruzione, potrete memorizzare il risultato della chiamata a **Thread.interrupted()**.

L'esempio che segue mostra la forma idiomatica tipica che dovrete utilizzare nel vostro metodo **run()** per gestire sia la possibilità di blocco sia quella di non blocco, quando è impostato lo stato di interrupt.

```
//: concurrency/InterruptingIdiom.java
// Forma idiomatica generale per l'interruzione di un task.
// {Args: 1100}
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class NeedsCleanup {
    private final int id;
    public NeedsCleanup(int ident) {
        id = ident;
        print("NeedsCleanup " + id);
    }
    public void cleanup() {
```



```
        print("Cleaning up " + id);
    }
}

class Blocked3 implements Runnable {
    private volatile double d = 0.0;
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // punto1
                NeedsCleanup n1 = new NeedsCleanup(1);
                // Attiva il blocco try-finally subito dopo la
                // definizione di n1, per assicurare il corretto
                // cleanup di n1:
                try {
                    print("Sleeping");
                    TimeUnit.SECONDS.sleep(1);
                    // punto2
                    NeedsCleanup n2 = new NeedsCleanup(2);
                    // Garantisce il corretto cleanup di n2:
                    try {
                        print("Calculating");
                        // Un'operazione non bloccante di lunga durata:
                        for(int i = 1; i < 2500000; i++)
                            d = d + (Math.PI + Math.E) / d;
                        print("Finished time-consuming operation");
                    } finally {
                        n2.cleanup();
                    }
                } finally {
                    n1.cleanup();
                }
            }
            print("Exiting via while() test");
        } catch(InterruptedException e) {
            print("Exiting via InterruptedException");
        }
    }
}
```



```
public class InterruptingIdiom {  
    public static void main(String[] args) throws Exception {  
        if(args.length != 1) {  
            print("usage: java InterruptingIdiom delay-in-mS");  
            System.exit(1);  
        }  
        Thread t = new Thread(new Blocked3());  
        t.start();  
        TimeUnit.MILLISECONDS.sleep(new Integer(args[0]));  
        t.interrupt();  
    }  
} /* Output: (Esempio)  
NeedsCleanup 1  
Sleeping  
NeedsCleanup 2  
Calculating  
Finished time-consuming operation  
Cleaning up 2  
Cleaning up 1  
NeedsCleanup 1  
Exiting via while()  
test  
*///:~
```

La classe **NeedsCleanup** enfatizza la necessità di un adeguato cleanup delle risorse, quando uscite dal ciclo mediante un'eccezione. Notate che tutte le risorse di **NeedsCleanup** create in **Blocked3.run()** devono essere immediatamente seguite dalle clausole **try-finally** per garantire che il metodo **cleanup()** venga sempre chiamato.

Dovete passare al programma un argomento da riga di comando, indicando i millisecondi di ritardo prima della chiamata a **interrupt()**. Con valori di ritardo differenti potrete uscire da **Blocked3.run()** in altri momenti del ciclo, nella chiamata **sleep()** bloccante e nel calcolo non bloccante. Vedrete che se è chiamato il metodo **interrupt()** dopo il commento “punto2” (durante l’operazione non bloccante), in primo luogo verrà completato il ciclo, poi saranno distrutti tutti gli oggetti locali, infine il ciclo verrà terminato nella sua parte superiore tramite l’istruzione **while**. Tuttavia, se la chiamata a **interrupt()** avviene tra “punto1” e “punto2”, vale a dire dopo l’istruzione **while** ma prima o nel corso dello **sleep()** che blocca l’operazione, il task terminerà per mezzo



dell'eccezione **InterruptedException**, la prima volta che si tenterà di eseguire un'operazione bloccante. In tal caso saranno sottoposti a cleanup soltanto gli oggetti **NeedsCleanup** creati fino al punto in cui l'eccezione sarà sollevata, e avrete così l'opportunità di eseguire qualsiasi altra operazione di cleanup nella clausola del **catch**.

Una classe progettata per rispondere a un **interrupt()** deve impostare una politica che le assicuri di rimanere in una condizione costante e coerente. Di norma questo significa che la creazione di tutti gli oggetti che richiedono il cleanup deve essere seguita dalle clausole **try-finally**, in modo che il cleanup avvenga a prescindere da come termina il ciclo **run()**. Un codice come questo può funzionare bene; purtroppo, però, il fatto che Java non preveda chiamate automatiche al distruttore fa sì che il programmatore client abbia il compito di scrivere le clausole **try-finally** adeguate.

## Cooperazione tra task

Come avete visto, quando utilizzate i thread per eseguire più task simultaneamente potete impedire che uno di questi interferisca con le risorse di un altro utilizzando un lock (mutex) per sincronizzare il comportamento dei due task. In pratica, se due task stanno per accedere a una risorsa condivisa, solitamente memoria, utilizzerete un mutex per consentire solo a un task alla volta di accedere a quella risorsa.

Dopo avere risolto questo problema, il passo successivo è imparare la tecnica per fare in modo che i task collaborino a vicenda, affinché diversi task possano cooperare alla risoluzione di un problema. In questo caso la questione non riguarda tanto la possibilità che i task interferiscano tra loro, quanto il modo per farli funzionare all'unisono, dal momento che alcune parti di un problema devono essere risolte prima di altre. Il concetto è analogo alla pianificazione di un progetto edilizio: in primo luogo occorre scavare le fondamenta di una costruzione, mentre i piloni e i muri portanti possono essere sviluppati in parallelo, ed entrambe queste attività devono essere terminate prima di poter procedere alla colata di cemento delle fondamenta. Le tubature idrauliche devono essere stese prima che la lastra di cemento armato possa essere colata, la lastra di cemento deve essere sul posto prima di iniziare a fissare le pareti e così via. Alcune di queste attività possono essere svolte in modo sincrono, ma determinate operazioni richiedono il completamento di altre fasi del progetto.

Il punto fondamentale da tenere presente in questo tipo di cooperazione è la necessità di impostare un canale di comunicazione tra i vari task, il cosiddetto *handshaking*, per implementare il quale si utilizza il mutex, in questo caso per garantire che soltanto un'operazione possa rispondere a un segnale.



Questo evita di incorrere nella corsa critica. In aggiunta al mutex, implementate un metodo per consentire a un task di autosospendersi in attesa del cambiamento di una condizione esterna (nel caso specifico, per esempio, "le tubature idrauliche ora sono a posto") che segnali al task quando è il momento di procedere oltre. In questi paragrafi si parlerà dell'handshaking tra i task, che viene implementato in modo sicuro utilizzando i metodi `wait()` e `notifyAll()` di `Object`. La libreria di concorrenza di Java SE5 mette a disposizione anche gli oggetti `Condition` con i metodi `await()` e `signal()`. Vedrete quali problemi possono presentarsi e come risolverli.

### ***Metodi wait() e notifyAll()***

Il metodo `wait()` consente di rimanere in attesa di un cambiamento in condizioni fuori del controllo del metodo corrente, che spesso vengono modificate da un altro task. Dovreste evitare che il vostro task iteri un "ciclo idle" per controllare il verificarsi della condizione; questo meccanismo è chiamato *busy wait* e di solito costituisce un esempio di cattivo impiego dei cicli di CPU, poiché, seppure in attesa, il task in realtà sta "consumando" CPU a discapito degli altri task, senza fare nulla di utile. Invece, `wait()` sospende il task mentre è in attesa del cambiamento della condizione bloccante: solo quando vengono chiamati `notify()` o `notifyAll()`, il che suggerisce che potrebbe essere avvenuto qualcosa di interessante, il task ritorna attivo per verificare gli eventuali cambiamenti. Quindi, il metodo `wait()` fornisce un meccanismo per sincronizzare le attività tra i task.

È importante comprendere che quando si chiama `sleep()` o `yield()` non viene rilasciato il lock sugli oggetti; invece, quando un'operazione entra in una chiamata a `wait()` all'interno di un metodo, l'esecuzione del thread viene sospesa e il lock sull'oggetto è rilasciato. Poiché `wait()` rilascia il lock, questo può essere acquisito da un altro task, in modo che altri metodi `synchronized` nell'oggetto (ora sbloccato) siano richiamabili durante `wait()`. Quest'ultima possibilità è essenziale, poiché questi altri metodi sono in genere all'origine del cambiamento di cui si è in attesa per la riattivazione del task sospeso. In pratica, quando chiamate `wait()` è come se diceste: "finora ho fatto tutto quanto potevo, per cui adesso mi limito ad aspettare, ma voglio consentire che altre operazioni `synchronized` possano avere luogo, se ciò è possibile".

Esistono due forme di `wait()`. Una variante accetta un argomento, espresso in millisecondi, che ha lo stesso significato dell'argomento di `sleep()`: "mettiti in pausa per il periodo di tempo indicato". Diversamente da `sleep()`, però, con `wait(pause)`:

1. il lock sull'oggetto viene rilasciato nel corso di `wait()`;
2. l'uscita da `wait()` avviene, oltre che all'esaurimento del tempo previsto, anche in seguito alle chiamate `notify()` e `notifyAll()`.



La seconda variante di `wait()`, di utilizzo più comune, non accetta argomenti: in questo caso `wait()` prosegue fino a quando il thread non riceve un messaggio `notify()` o `notifyAll()`.

Un aspetto peculiare di `wait()`, `notify()` e `notifyAll()` è che questi metodi fanno parte della classe di base `Object`, non di `Thread`. Anche se a prima vista la presenza di funzionalità dedicate al threading nella classe di base universale può sembrare strana, è una condizione essenziale perché questi metodi gestiscono il lock, che è anch'esso parte di ogni oggetto. Di conseguenza potete includere `wait()` in qualsiasi metodo `synchronized`, indipendentemente dal fatto che tale classe estenda `Thread` o implementi `Runnable`. Infatti, l'unico punto in cui potete chiamare `wait()`, `notify()` o `notifyAll()` è all'interno di un metodo o blocco `synchronized`; di contro, il metodo `sleep()` può essere chiamato nell'ambito di metodi non `synchronized` poiché non gestisce il lock. Se chiamate uno di questi metodi all'interno di un metodo non `synchronized` il programma compilerà regolarmente, ma quando lo eseguirete otterrete un'eccezione `IllegalMonitorStateException`, accompagnata dal messaggio, decisamente poco intuitivo, "current thread not owner": questo sta a indicare che il task che chiama `wait()`, `notify()` o `notifyAll()`, prima di chiamare questi metodi deve "possedere" (acquisire) il lock per l'oggetto.

Potete chiedere a un altro oggetto di eseguire un'operazione che manipola il suo lock; a questo scopo dovete innanzitutto intercettare il lock di quell'oggetto. Per esempio, se desiderate trasmettere `notifyAll()` a un oggetto `x` dovreste farlo all'interno di un blocco `synchronized` che acquisisca il lock per `x`:

```
synchronized(x) {  
    x.notifyAll();  
}
```

Considerate un semplice esempio. **WaxOMatic.java** ha due processi: uno per la ceratura di un veicolo (**Car**) e uno per la lucidatura. Il task di lucidatura non può eseguire il lavoro fino a quando quello di ceratura non sia terminato, e il task di ceratura deve attendere il termine del task di lucidatura prima di applicare un nuovo strato di cera. Sia **WaxOn** sia **WaxOff** utilizzano l'oggetto **Car**, il quale a sua volta si serve dei metodi `wait()` e `notifyAll()` per sospendere i task e riavivarli mentre sono in attesa del cambiamento di una condizione.

```
//: concurrency/waxomatic/WaxOMatic.java  
// Cooperazione di base tra task.  
package concurrency.waxomatic;
```



```
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Car {
    private boolean waxOn = false;
    public synchronized void waxed() {
        waxOn = true; // Pronto per la lucidatura
        notifyAll();
    }
    public synchronized void buffed() {
        waxOn = false; // Pronto per un altro strato di cera
        notifyAll();
    }
    public synchronized void waitForWaxing()
        throws InterruptedException {
        while(waxOn == false)
            wait();
    }
    public synchronized void waitForBuffing()
        throws InterruptedException {
        while(waxOn == true)
            wait();
    }
}

class WaxOn implements Runnable {
    private Car car;
    public WaxOn(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                printnb("Wax On! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.waxed();
                car.waitForBuffing();
            }
        } catch(InterruptedException e) {
```



```
        print("Exiting via interrupt");
    }
    print("Ending Wax On task");
}
}

class WaxOff implements Runnable {
    private Car car;
    public WaxOff(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                car.waitForWaxing();
                println("Wax Off! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.buffed();
            }
        } catch(InterruptedException e) {
            print("Exiting via interrupt");
        }
        print("Ending Wax Off task");
    }
}

public class WaxOMatic {
    public static void main(String[] args) throws Exception {
        Car car = new Car();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new WaxOff(car));
        exec.execute(new WaxOn(car));
        TimeUnit.SECONDS.sleep(5); // Disattiva per il tempo
                                // indicato
        exec.shutdownNow(); // Interrompe tutti i task
    }
} /* Output: (95% match)
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off!
```



```

Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Exiting via interrupt
Ending Wax On task
Exiting via interrupt
Ending Wax Off task
*///:~

```

In questo esempio, **Car** ha una variabile **boolean waxOn** che indica la condizione del processo di ceratura e lucidatura.

Il metodo **waitForWaxing()** verifica il flag **waxOn**: se è **false**, il task chiamante viene sospeso chiamando **wait()**. È importante che questo avvenga in un metodo **synchronized**, dove il task ha acquisito il lock. Quando chiamate **wait()** il thread viene sospeso e il lock rilasciato; è essenziale che il lock sia rilasciato poiché, al fine di modificare in modo sicuro lo stato dell'oggetto (per esempio, cambiare **waxOn** in **true**, come dovrebbe accadere se il task sospeso dovesse continuare), il lock deve essere disponibile all'acquisizione da parte di un altro task. In questo esempio, quando altri task chiamano **waxed()** per indicare che è il momento di eseguire qualcosa, deve essere acquisito il lock per cambiare **waxOn** in **true**. Poi **waxed()** chiama **notifyAll()**, che riattiva il task sospeso dalla chiamata a **wait()**. Per fare in modo che il task si riattivi da **wait()** per prima cosa esso deve riacquisire il lock che aveva rilasciato al momento di entrare in **wait()**. Il task non potrà riattivarsi fino a quando il lock non sarà disponibile.<sup>16</sup>

Il metodo **WaxOn.run()** rappresenta il primo passo della ceratura dell'automobile, pertanto esegue il suo compito sotto forma di una chiamata a **sleep()** che simula il tempo necessario per la ceratura del veicolo; poi segnala al veicolo (**car**) che l'operazione è completa e chiama **waitForBuffing()**, che sospende questa operazione con **wait()** fino a quando il task **WaxOff** chiama **buffed()** per l'oggetto **car**, modificando lo stato e chiamando **notifyAll()**. Invece **WaxOff.run()** entra immediatamente in **waitForWaxing()**, poi viene sospeso fino all'applicazione della cera tramite **WaxOn** e alla chiamata al metodo **waxed()**. Eseguendo il programma noterete che questo processo a

---

16. Alcune piattaforme prevedono un terzo modo per uscire da **wait()**: il cosiddetto “*wake-up spurious*”. In pratica, l'espressione “risveglio spurio” significa che un thread può uscire in anticipo dal suo stato di blocco, mentre è in attesa della modifica di uno stato o di un semaforo, senza essere richiamato esplicitamente da **notify()** o **notifyAll()** o dai loro equivalenti per i nuovi oggetti **Condition**: il thread si riattiva, all'apparenza da sé. I wake-up spuri esistono perché su alcune piattaforme l'implementazione dei thread secondo lo standard POSIX, o equivalente, non è sempre immediata come dovrebbe essere. L'esistenza di questa funzionalità semplifica su tali piattaforme la costruzione di una libreria come pthread.



due fasi si ripete, mentre il controllo viene alternato tra i due task. Dopo cinque secondi i thread sono entrambi arrestati con `interrupt()`: infatti, quando chiamate `shutdownNow()` per `ExecutorService`, esso chiama `interrupt()` per tutti i task che sta gestendo.

L'esempio precedente evidenzia la necessità di racchiudere la chiamata a `wait()` in un ciclo `while` per verificare lo stato (o gli statii) da controllare. Questo è importante per i motivi elencati di seguito.

1. Potrebbero sussistere numerosi task in attesa sullo stesso lock e per lo stesso motivo, e il primo task che si riattiva potrebbe modificare la situazione: non dimenticate che se anche non implementate voi stessi una situazione simile, qualcun altro potrebbe ereditare dalla vostra classe e farlo. Se è il caso, questo task dovrebbe essere sospeso finché non venga modificata la relativa condizione di interesse.
2. Prima che il task si riprenda da `wait()` è possibile che altri task abbiano modificato la situazione, in modo tale che il task non sia in grado o non abbia interesse a eseguire le proprie operazioni. Anche in questo caso, esso dovrebbe essere nuovamente sospeso chiamando `wait()`.
3. È anche possibile che i task rimangano in attesa sul lock del vostro oggetto per motivi diversi, nel qual caso dovrete utilizzare `notifyAll()`. In queste situazioni dovrete controllare se la ripresa è avvenuta per motivi legittimi: in caso contrario chiamerete di nuovo `wait()`.

È quindi essenziale che controlliate la particolare condizione di interesse, tornando in `wait()` qualora tale condizione non venga soddisfatta. Tutto questo viene sintetizzato nella forma idiomatica `while`.

**Esercizio 21** (2) Create due classi di tipo `Runnable`, la prima con un metodo `run()` che si avvia e chiama `wait()`; la seconda classe dovrà gestire il riferimento del primo oggetto `Runnable`. Il metodo `run()` del secondo oggetto deve chiamare `notifyAll()` per il primo task, dopo un certo numero di secondi, in modo che questo possa visualizzare un messaggio. Verificate le vostre classi utilizzando un `Executor`.

**Esercizio 22** (4) Create un esempio di *busy wait*. Un task rimane in `sleep()` per qualche tempo prima di impostare un flag a `true`. Il secondo task monitora questo flag all'interno di un ciclo `while` (il *busy wait*, appunto); quando il flag diventa `true`, lo reimposta a `false` e visualizza la modifica a console. Prendete nota del tempo sprecato dal programma all'interno del ciclo e create una seconda versione di questo progetto che utilizzi il metodo `wait()` anziché il meccanismo *busy wait*.



## Segnali mancati

Quando due thread vengono coordinati utilizzando **notify()**/**wait()** o **notifyAll()**/**wait()** è possibile che manchi un segnale. Supponete che **T1** sia un thread che esegue notifiche a **T2** e che i due thread siano implementati in forma errata.

```
T1:  
synchronized(sharedMonitor) {  
    <impostazione condizione per T2>  
    sharedMonitor.notify();  
}  
  
T2:  
while(someCondition) {  
    // Punto 1  
    synchronized(sharedMonitor) {  
        sharedMonitor.wait();  
    }  
}
```

Il codice che corrisponde a **<impostazione condizione per T2>** descrive un'azione che impedisce a **T2** di chiamare **wait()**.

Supponete che **T2** valuti la variabile **someCondition** e la trovi **true**. In corrispondenza del commento **Punto 1** il pianificatore di thread potrebbe passare a **T1**. A quel punto **T1** eseguirà l'impostazione prevista, poi chiamerà **notify()**. Quando **T2** continuerà l'esecuzione sarà troppo tardi perché si renda conto che nel frattempo la condizione è cambiata, pertanto entrerà in **wait()**. Il metodo **notify()** sarà ignorato e **T2** aspetterà per un tempo illimitato un segnale che è già stato trasmesso, producendo così una condizione di deadlock.

La soluzione consiste nell'impedire la corsa critica per la variabile **someCondition**. Ecco l'approccio corretto per **T2**:

```
synchronized(sharedMonitor) {  
    while(someCondition)  
        sharedMonitor.wait();  
}
```



Ora, se **T1** viene eseguito per primo, quando il controllo ritorna di nuovo a **T2** questo thread si renderà conto che la condizione è cambiata e non entrerà in `wait()`. Al contrario, se è **T2** a essere eseguito per primo, entrerà in `wait()` e successivamente sarà riattivato dal thread **T1**. In questo modo il segnale verrà comunque intercettato.

### **Confronto tra `notify()` e `notifyAll()`**

Poiché tecnicamente più di un task potrebbe essere in `wait()` su uno stesso oggetto **Car**, è più sicuro chiamare `notifyAll()` invece di `notify()`. Tuttavia la struttura del programma precedente è tale che soltanto un task sarà in `wait()`, pertanto potrete utilizzare `notify()` anziché `notifyAll()`.

L'utilizzo di `notify()` in luogo di `notifyAll()` è una forma di ottimizzazione. Dei numerosi task che potrebbero essere in attesa su un lock soltanto uno verrà riattivato con `notify()`, quindi, se volete utilizzare questo metodo dovete accertarvi che venga riattivato il task corretto. Inoltre, l'utilizzo di `notify()` richiede che tutti i task siano in attesa della stessa condizione, dal momento che se avete task che stanno attendendo condizioni differenti non sapete se si riattiverà quello giusto: per utilizzare `notify()`, una sola operazione deve beneficiare del cambio di condizione. Infine, questi vincoli devono sempre essere validi per tutte le sottoclassi possibili: se una qualsiasi di queste regole non potrà essere soddisfatta dovrete ricorrere a `notifyAll()` invece che a `notify()`.

Una delle affermazioni più confuse che ricorrono nelle discussioni sul threading Java è il fatto che `notifyAll()` “riattiva tutti i task in attesa”. Vorrebbe forse dire che qualsiasi task sia in `wait()`, in qualsiasi punto del programma, viene riattivato da qualunque chiamata a `notifyAll()`? Nell'esempio seguente il codice associato a **Task2** indica che questo non è vero. Infatti, solo i task che sono in attesa su un particolare lock vengono riattivati chiamando `notifyAll()` per quel lock.

```
//: concurrency/NotifyVsNotifyAll.java
import java.util.concurrent.*;
import java.util.*;

class Blocker {
    synchronized void waitingCall() {
        try {
            while(!Thread.interrupted()) {
                wait();
            }
        } catch(InterruptedException ie) {
            Thread.currentThread().interrupt();
        }
    }
}
```



```
        System.out.print(Thread.currentThread() + " ");
    }
} catch(InterruptedException e) {
    // OK per uscire in questo modo
}
}

synchronized void prod() { notify(); }
synchronized void prodAll() { notifyAll(); }
}

class Task implements Runnable {
    static Blocker blocker = new Blocker();
    public void run() { blocker.waitingCall(); }
}

class Task2 implements Runnable {
    // Un oggetto Blocker separato:
    static Blocker blocker = new Blocker();
    public void run() { blocker.waitingCall(); }
}

public class NotifyVsNotifyAll {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new Task());
        exec.execute(new Task2());
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(new TimerTask() {
            boolean prod = true;
            public void run() {
                if(prod) {
                    System.out.print("\nnotify() ");
                    Task.blocker.prod();
                    prod = false;
                } else {
                    System.out.print("\nnotifyAll() ");
                }
            }
        }, 0, 1000);
    }
}
```



```
    Task.blocker.prodAll();
    prod = true;
}
}
}, 400, 400); // Viene eseguito ogni 0,4 secondi
TimeUnit.SECONDS.sleep(5); // Disattiva per il tempo
                           // indicato
timer.cancel();
System.out.println("\nTimer cancelled");
TimeUnit.MILLISECONDS.sleep(500);
System.out.print("Task2.blocker.prodAll() ");
Task2.blocker.prodAll();
TimeUnit.MILLISECONDS.sleep(500);
System.out.println("\nShutting down");
exec.shutdownNow(); // Interrompe tutti i task
}
} /* Output: (Esempio)
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main]
Thread[pool-1-thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main]
Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main]
Thread[pool-1-thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main]
Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main]
Thread[pool-1-thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-2,5,main]
```



```
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main]
Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-5,5,main]
Timer canceled
Task2.blocker.prodAll() Thread[pool-1-thread-6,5,main]
Shutting down
*///:~
```

Ciascuna delle classi **Task** e **Task2** ha il suo oggetto **Blocker**, pertanto ogni oggetto **Task** si bloccerà su **Task.blocker**, e ogni oggetto **Task2** si bloccerà su **Task2.blocker**. In **main()** un oggetto **java.util.Timer** viene impostato per eseguire il suo metodo **run()** ogni 4/10 di secondo, e tale **run()** alterna le chiamate **notify()** e **notifyAll()** su **Task.blocker** tramite i metodi “prod”.

Dall'output potete notare che anche se un oggetto **Task2** esiste ed è bloccato su **Task2.blocker**, nessuna delle chiamate **notify()** o **notifyAll()** su **Task.blocker** provocherà la riattivazione dell'oggetto **Task2**. Allo stesso modo, al termine di **main()** viene chiamato il metodo **timer.cancel()**: anche se il timer è annullato, i primi cinque task rimangono attivi e bloccati nelle rispettive chiamate a **Task.blocker.waitingCall()**. L'output di **Task2.blocker.prodAll()** non include nessuno dei task che sono in attesa sul lock in **Task.blocker**.

Anche questo è perfettamente logico, tenuto conto dei metodi **prod()** e **prodAll()** in **Blocker**. Si tratta di metodi **synchronized**, il che significa che acquisiscono il loro lock, pertanto quando chiamano **notify()** o **notifyAll()** è logico che la chiamata sia soltanto per quel lock, per riattivare i task che stanno attendendo sul lock specifico.

La chiamata a **Blocker.waitingCall()** è così semplice che, in questo caso, potrebbe essere resa nel ciclo **for** come in un'istruzione **while(!Thread.interrupted())**, ottenendo lo stesso effetto: in questo esempio, infatti, è indifferente uscire dal ciclo con un'eccezione o lasciarlo controllando il flag **interrupted()**, dal momento che viene eseguito lo stesso codice in entrambi i casi. Per questioni di forma, però, l'esempio controlla **interrupted()**, poiché ci sono due modi differenti per lasciare il ciclo. Qualora in seguito decidiate di aggiungere codice al ciclo, rischiereste di introdurre un errore non coprendo entrambi i percorsi dell'output dal ciclo stesso.

**Esercizio 23 (7)** Dimostrate che **WaxOMatic.java** funziona correttamente quando utilizzate **notify()** invece di **notifyAll()**.



## Produttori e consumatori

Considerate un ristorante che abbia uno chef e un solo addetto ai tavoli, che naturalmente deve aspettare i vari piatti preparati dallo chef. Quando lo chef ha una portata pronta informa il cameriere, che preleva, trasporta il piatto e si rimette in attesa. Questo è un esempio della cooperazione tra task: lo chef ha il ruolo di *produttore* e il cameriere rappresenta il *consumatore*. Entrambi i task devono essere in comunicazione (*handshaking*) mentre le portate vengono prodotte e consumate, e il sistema deve interrompere le operazioni in modo ordinato. Il modello tradotto in codice è elencato di seguito.

```
//: concurrency/Restaurant.java
// L'approccio produttore-consumatore alla cooperazione tra i
// task.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Meal {
    private final int orderNum;
    public Meal(int orderNum) { this.orderNum = orderNum; }
    public String toString() { return "Meal " + orderNum; }
}

class WaitPerson implements Runnable {
    private Restaurant restaurant;
    public WaitPerson(Restaurant r) { restaurant = r; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    while(restaurant.meal == null)
                        wait(); // ...per dare il tempo allo chef
                                // di preparare il piatto
                }
                print("Waitperson got " + restaurant.meal);
                synchronized(restaurant.chef) {
                    restaurant.meal = null;
                }
            }
        } catch(InterruptedException e) {
            System.out.println("Waitperson interrupted");
        }
    }
}
```



```
        restaurant.chef.notifyAll(); // Pronto per un altro
                                     // piatto
    }
}
} catch(InterruptedException e) {
    print("WaitPerson interrupted");
}
}
}

class Chef implements Runnable {
    private Restaurant restaurant;
    private int count = 0;
    public Chef(Restaurant r) { restaurant = r; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    while(restaurant.meal != null)
                        wait(); // ...per dare il tempo al cameriere
                                // di prendere il piatto
                }
                if(++count == 10) {
                    print("Out of food, closing");
                    restaurant.exec.shutdownNow();
                }
                printnb("Order up! ");
                synchronized(restaurant.waitPerson) {
                    restaurant.meal = new Meal(count);
                    restaurant.waitPerson.notifyAll();
                }
                TimeUnit.MILLISECONDS.sleep(100);
            }
        } catch(InterruptedException e) {
            print("Chef interrupted");
        }
    }
}
```



```
public class Restaurant {  
    Meal meal;  
    ExecutorService exec = Executors.newCachedThreadPool();  
    WaitPerson waitPerson = new WaitPerson(this);  
    Chef chef = new Chef(this);  
    public Restaurant() {  
        exec.execute(chef);  
        exec.execute(waitPerson);  
    }  
    public static void main(String[] args) {  
        new Restaurant();  
    }  
} /* Output:  
Order up! Waitperson got Meal 1  
Order up! Waitperson got Meal 2  
Order up! Waitperson got Meal 3  
Order up! Waitperson got Meal 4  
Order up! Waitperson got Meal 5  
Order up! Waitperson got Meal 6  
Order up! Waitperson got Meal 7  
Order up! Waitperson got Meal 8  
Order up! Waitperson got Meal 9  
Out of food, closing  
WaitPerson interrupted  
Order up! Chef interrupted  
*///:~
```

**Restaurant** è il punto focale sia per **WaitPerson** sia per **Chef**: entrambi devono conoscere per quale **Restaurant** stanno lavorando, poiché devono mettere a disposizione o prelevare il piatto dal "passavivande" del ristorante, **restaurant.meal**. In **run()** il cameriere (**WaitPerson**) entra in modalità **wait()**, interrompendo quel task finché non venga riattivato da un **notifyAll()** dello **Chef**. Trattandosi di un programma molto semplice, si è fatto in modo che soltanto un task sia in attesa sul lock di **WaitPerson**: il task **WaitPerson** stesso. Per questo motivo è teoricamente possibile chiamare **notify()** anziché **notifyAll()**. Tuttavia, in situazioni più complesse diversi task potrebbero rimanere in attesa su un lock particolare dell'oggetto, di conseguenza non sapreste quale task dovrebbe essere riattivato. È quindi più sicuro chiamare **notifyAll()**, che riattiva tutti i task in attesa su quel lock: sarà poi il task stesso a decidere se la notifica gli compete.



Una volta che lo **Chef** ha fornito un **Meal** e informato il cameriere **WaitPerson**, lo **Chef** attende finché **WaitPerson** non avrà prelevato il piatto informandone lo **Chef**, che potrà così produrre il **Meal** seguente. Notate che **wait()** è incorporato in un'istruzione **while()** che sta testando la stessa condizione di cui è in attesa. All'apparenza questo sembra piuttosto strano: in effetti, come camerieri, se state aspettando un'ordinazione vi aspettereste che l'ordine sia disponibile quando venite richiamati. Come si è detto in precedenza, il problema è che in un'applicazione concorrente un altro task potrebbe "intrufolarsi" e servire l'ordinazione mentre **WaitPerson** si sta riattivando. L'unico approccio sensato è utilizzare sempre il seguente idioma per un **wait()**, naturalmente con l'opportuna sincronizzazione e tenendo conto, nel programma, della possibilità di mancati segnali:

```
while(conditionIsNotMet)  
    wait();
```

Questo garantisce che la condizione sia soddisfatta prima di uscire dal ciclo di attesa, e se siete stati informati di qualcosa che non si riferisce alla condizione (come può accadere con **notifyAll()**), o se la condizione cambia prima che usciate completamente dal ciclo di attesa, è certo che ritornerete nuovamente in attesa.

Notate che la chiamata a **notifyAll()** deve prima intercettare il lock su **WaitPerson**. La chiamata a **wait()** in **WaitPerson.run()** rilascia automaticamente il lock, pertanto tale eventualità è possibile. Poiché il lock deve essere posseduto affinché **notifyAll()** venga chiamato, è certo che due task che provano a chiamare **notifyAll()** su uno stesso oggetto non si intralceranno a vicenda.

Entrambi i metodi **run()** sono progettati per provocare un arresto ordinato incorporando l'intero **run()** in un blocco **try**. La clausola **catch** si chiude immediatamente prima della parentesi di chiusura del metodo **run()**, in modo che se il task riceve un'**InterruptedException** si conclude subito dopo avere intercettato l'eccezione.

Tenete presente che, in **Chef**, dopo avere chiamato **shutdownNow()** potreste semplicemente ritornare (**return**) da **run()**, e normalmente è proprio quello che dovreste fare. Tuttavia è più interessante farlo nel modo indicato nel codice. Ricordate che **shutdownNow()** invia un **interrupt()** a tutti i task avviati da **ExecutorService**. Nel caso di **Chef** però il task non si interrompe immediatamente su **interrupt()**, poiché questo solleva un'**InterruptedException** soltanto quando il task cerca di accedere a un'operazione bloccante (interrompibile). Pertanto sarà visualizzato innanzitutto il messaggio "Avanti con gli ordini!", poi verrà sollevata l'**InterruptedException** non appena **Chef** cercherà di chia-



mare `sleep()`. Se rimuovete la chiamata a `sleep()` il task arriverà alla parte superiore del ciclo `run()` e terminerà a causa del test `Thread.interrupted()`, senza sollevare un'eccezione.

Nell'esempio precedente vi è soltanto un punto in cui un task può memorizzare un oggetto che sia poi utilizzabile da un altro task. Di norma, tuttavia, in un'implementazione di tipo produttore-consomatore vi servirete di una coda FIFO per memorizzare gli oggetti prodotti e consumati. L'argomento delle code sarà esaminato nel prosieguo del capitolo.

**Esercizio 24** (1) Risolvete un problema di tipo “unico produttore, unico consumatore” utilizzando `wait()` e `notifyAll()`. Il produttore non dovrà eccedere la capacità di ricezione del buffer ricevente, situazione che può verificarsi se il produttore è più veloce del consumatore; nel caso opposto, invece, non dovrà leggere più di una volta gli stessi dati. Non date per scontato nulla riguardo alle velocità relative al produttore e al consumatore.

**Esercizio 25** (1) Nella classe `Chef` in `Restaurant.java` ritornate (`return`) da `run()` dopo avere chiamato `shutdownNow()`, e osservate la differenza di comportamento.

**Esercizio 26** (8) Aggiungete una classe `BusBoy` a `Restaurant.java`. Dopo che il piatto è stato consegnato, `WaitPerson` dovrebbe avvisare `BusBoy` di pulire il tavolo.

#### *Utilizzo di oggetti Lock e Condition esplicativi*

La libreria `java.util.concurrent` offre strumenti supplementari ed esplicativi utilizzabili per la riscrittura di `WaxOMatic.java`. La classe di base che utilizza un mutex e consente la sospensione dei task è `Condition`, ed è possibile sospendere un'operazione chiamando `await()` su un oggetto `Condition`. Quando si verificano cambiamenti di stato provenienti dall'esterno e tali da indicare che un task dovrebbe continuare la propria elaborazione, potete informarne il task chiamando il metodo `signal()` per riattivarlo, oppure chiamare `signalAll()` per riattivare tutti i task sospesi su quell'oggetto `Condition`. Tenete presente che, come `notifyAll()`, il metodo `signalAll()` è l'approccio più sicuro.

Di seguito è mostrata la nuova versione del programma `WaxOMatic.java`, riscritto per contenere una `Condition` utilizzata per sospendere un task all'interno di `waitForWaxing()` o `waitForBuffing()`.

```
//: concurrency/waxomatic2/WaxOMatic2.java
// Utilizzo di oggetti Lock e Condition.
package concurrency.waxomatic2;
```



```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class Car {
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();
    private boolean waxOn = false;
    public void waxed() {
        lock.lock();
        try {
            waxOn = true; // Pronto per la lucidatura
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
    public void buffed() {
        lock.lock();
        try {
            waxOn = false; // Pronto per un altro strato di cera
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
    public void waitForWaxing() throws InterruptedException {
        lock.lock();
        try {
            while(waxOn == false)
                condition.await();
        } finally {
            lock.unlock();
        }
    }
    public void waitForBuffing() throws InterruptedException{
        lock.lock();
```



```
try {
    while(waxOn == true)
        condition.await();
} finally {
    lock.unlock();
}
}

class WaxOn implements Runnable {
    private Car car;
    public WaxOn(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                printnb("Wax On! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.waxed();
                car.waitForBuffing();
            }
        } catch(InterruptedException e) {
            print("Exiting via interrupt");
        }
        print("Ending Wax On task");
    }
}

class WaxOff implements Runnable {
    private Car car;
    public WaxOff(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                car.waitForWaxing();
                printnb("Wax Off! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.buffed();
            }
        }
    }
}
```



```
        } catch(InterruptedException e) {
            print("Exiting via interrupt");
        }
        print("Ending Wax Off task");
    }
}

public class WaxOMatic2 {
    public static void main(String[] args) throws Exception {
        Car car = new Car();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new WaxOff(car));
        exec.execute(new WaxOn(car));
        TimeUnit.SECONDS.sleep(5);
        exec.shutdownNow();
    }
} /* Output: (90% match)
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off!
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Exiting via interrupt
Ending Wax Off task
Exiting via interrupt
Ending Wax On task
*/://:~
```

Nel costruttore di **Car** un unico **Lock** produce un oggetto **Condition** che è utilizzato per gestire la comunicazione tra i task. Tuttavia l'oggetto **Condition** non contiene i dati sullo stato del processo, ed è quindi necessario gestire informazioni supplementari per segnalare tale stato, che in questo caso è rappresentato dalla variabile **boolean waxOn**.

Ogni chiamata a **lock()** deve essere immediatamente seguita dal blocco **try-finally** per garantire che lo sbloccaggio avvenga in qualsiasi caso. Come con le versioni native, prima di potere chiamare **await()**, **signal()** o **signalAll()** un task deve possedere il lock.

Notate che questa soluzione è più complessa della precedente, una complessità che nel caso specifico non porta alcun vantaggio. Gli oggetti **Condition** e **Lock** sono richiesti soltanto per problemi di threading più complessi.



Esercizio 27 (2) Modificate **Restaurant.java** affinché utilizzi gli oggetti **Condition** e **Lock** espliciti.

### **Produttori, consumatori e code**

I metodi **wait()** e **notifyAll()** risolvono il problema della cooperazione tra i task a un livello alquanto basso, sincronizzando le interazioni. In molti casi potete astrarre di un livello e risolvere i problemi di cooperazione tra i task utilizzando una *coda sincronizzata* (*synchronized queue*), che consente a un solo task per volta di inserire o rimuovere un elemento. La coda sincronizzata è disponibile nell'interfaccia **java.util.concurrent.BlockingQueue**, di cui esiste un certo numero di implementazioni standard. Di norma utilizzerete **LinkedBlockingQueue**, che è una coda di dimensioni illimitate; **ArrayBlockingQueue** ha invece dimensioni fisse, quindi potete inserirvi soltanto un certo numero di elementi prima che vada in blocco.

Inoltre queste code sospendono un task consumatore se questo cerca di ottenere un oggetto da una coda vuota, riattivandolo non appena altri elementi diventano disponibili. Il bloccaggio delle code può risolvere molti problemi in modo più semplice e sicuro dei metodi **wait()** e **notifyAll()**.

Ecco un semplice esempio che serializza l'esecuzione di oggetti **LiftOff**. Il consumatore è **LiftOffRunner**, che gestisce ogni oggetto **LiftOff** fuori da **BlockingQueue** e lo esegue direttamente: in pratica utilizza il proprio thread chiamando **run()**, invece di avviare un nuovo thread per ogni task.

```
//: concurrency/TestBlockingQueues.java
// {RunByHand}
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class LiftOffRunner implements Runnable {
    private BlockingQueue<LiftOff> rockets;
    public LiftOffRunner(BlockingQueue<LiftOff> queue) {
        rockets = queue;
    }
    public void add(LiftOff lo) {
        try {
            rockets.put(lo);
        } catch(InterruptedException e) {

```



```
        print("Interrupted during put()");
    }
}

public void run() {
    try {
        while(!Thread.interrupted()) {
            LiftOff rocket = rockets.take();
            rocket.run(); // utilizzate questo thread
        }
    } catch(InterruptedException e) {
        print("Waking from take()");
    }
    print("Exiting LiftOffRunner");
}
}

public class TestBlockingQueues {
    static void getkey() {
        try {
            // Compensa la differenza tra Windows e Linux nella
            // lunghezza del risultato prodotto dal tasto Invio:
            new BufferedReader(
                new InputStreamReader(System.in)).readLine();
        } catch(java.io.IOException e) {
            throw new RuntimeException(e);
        }
    }
    static void getkey(String message) {
        print(message);
        getkey();
    }
    static void
    test(String msg, BlockingQueue<LiftOff> queue) {
        print(msg);
        LiftOffRunner runner = new LiftOffRunner(queue);
        Thread t = new Thread(runner);
        t.start();
    }
}
```



```
for(int i = 0; i < 5; i++)
    runner.add(new LiftOff(5));
getkey("Press 'Enter' (" + msg + ")");
t.interrupt();
print("Finished " + msg + " test");
}
public static void main(String[] args) {
    test("LinkedBlockingQueue", // Dimensioni illimitate
        new LinkedBlockingQueue<LiftOff>());
    test("ArrayBlockingQueue", // Dimensioni fisse
        new ArrayBlockingQueue<LiftOff>(3));
    test("SynchronousQueue", // Dimensioni pari a 1
        new SynchronousQueue<LiftOff>());
}
} //:~
```

I task sono inseriti nella coda **BlockingQueue** da **main()** ed eliminati da essa per mezzo di **LiftOffRunner**. Notate che **LiftOffRunner** può ignorare i problemi di sincronizzazione poiché risolti dalla **BlockingQueue**.

**Esercizio 28** (3) Modificate **TestBlockingQueues.java**: l'inserimento di **LiftOff** nella **BlockingQueue** dovrà avvenire con un nuovo task, invece che direttamente in **main()**.

### Utilizzo di **BlockingQueue** nella produzione di toast!

Come esempio dell'utilizzo di **BlockingQueue** immaginate una macchina che esegue tre attività, o task: la tostatura delle fette di pane, la loro imburratura e la spalmatura di marmellata. È possibile produrre il toast completo ricorrendo alle **BlockingQueue**, come mostrato di seguito.

```
//: concurrency/ToastOMatic.java
// Un produttore di toast che utilizza le code.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

```
class Toast {
    public enum Status { DRY, BUTTERED, JAMMED }
```



```
private Status status = Status.DRY;
private final int id;
public Toast(int idn) { id = idn; }
public void butter() { status = Status.BUTTERED; }
public void jam() { status = Status.JAMMED; }
public Status getStatus() { return status; }
public int getId() { return id; }
public String toString() {
    return "Toast " + id + ":" + status;
}
}

class ToastQueue extends LinkedBlockingQueue<Toast> {}

class Toaster implements Runnable {
    private ToastQueue toastQueue;
    private int count = 0;
    private Random rand = new Random(47);
    public Toaster(ToastQueue tq) { toastQueue = tq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(
                    100 + rand.nextInt(500));
                // Prepara il pane tostato
                Toast t = new Toast(count++);
                print(t);
                // Inserisce nella coda
                toastQueue.put(t);
            }
        } catch(InterruptedException e) {
            print("Toaster interrupted");
        }
        print("Toaster off");
    }
}
```



```
// Applica il burro sul toast:  
class Butterer implements Runnable {  
    private ToastQueue dryQueue, butteredQueue;  
    public Butterer(ToastQueue dry, ToastQueue buttered) {  
        dryQueue = dry;  
        butteredQueue = buttered;  
    }  
    public void run() {  
        try {  
            while(!Thread.interrupted()) {  
                // Si blocca finche' non e' disponibile un'altra fetta  
                // di pane tostato:  
                Toast t = dryQueue.take();  
                t.butter();  
                print(t);  
                butteredQueue.put(t);  
            }  
        } catch(InterruptedException e) {  
            print("Butterer interrupted");  
        }  
        print("Butterer off");  
    }  
}  
  
// Applica la marmellata sul toast imburrato:  
class Jammer implements Runnable {  
    private ToastQueue butteredQueue, finishedQueue;  
    public Jammer(ToastQueue buttered, ToastQueue finished) {  
        butteredQueue = buttered;  
        finishedQueue = finished;  
    }  
    public void run() {  
        try {  
            while(!Thread.interrupted()) {  
                // Si blocca finche' non e' disponibile un'altra fetta  
                // di pane imburrato  
                Toast t = butteredQueue.take();
```



```
        t.jam();
        print(t);
        finishedQueue.put(t);
    }
} catch(InterruptedException e) {
    print("Jammer interrupted");
}
print("Jammer off");
}
}

// Consuma il toast:
class Eater implements Runnable {
    private ToastQueue finishedQueue;
    private int counter = 0;
    public Eater(ToastQueue finished) {
        finishedQueue = finished;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Si blocca finche' non e' disponibile un altro
                // toast:
                Toast t = finishedQueue.take();
                // Verifica che i toast siano prodotti in ordine,
                // e che tutti abbiano la marmellata:
                if(t.getId() != counter++) ||
                    t.getStatus() != Toast.Status.JAMMED) {
                    print(">>> Error: " + t);
                    System.exit(1);
                } else
                    print("Chomp! " + t);
            }
        } catch(InterruptedException e) {
            print("Eater interrupted");
        }
    }
}
```



```
    print("Eater off");
}
}

public class ToastOMatic {
    public static void main(String[] args) throws Exception {
        ToastQueue dryQueue = new ToastQueue(),
                    butteredQueue = new ToastQueue(),
                    finishedQueue = new ToastQueue();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new Toaster(dryQueue));
        exec.execute(new Butterer(dryQueue, butteredQueue));
        exec.execute(new Jammer(butteredQueue, finishedQueue));
        exec.execute(new Eater(finishedQueue));
        TimeUnit.SECONDS.sleep(5);
        exec.shutdownNow();
    }
} /* (Da eseguire per visualizzare l'output) *///:~
```

**Toast** è un eccellente esempio del valore delle **enum**. Notate che non vi è sincronizzazione esplicita, ottenuta con oggetti **Lock** o mediante la parola chiave **synchronized**, poiché la sincronizzazione è gestita implicitamente dalle code, sincronizzate internamente, e a livello progettuale dal sistema: ogni **Toast** è gestito solo da un task per volta.

Poiché la coda è bloccante i processi si interrompono e riprendono automaticamente. Come potete notare, la semplificazione prodotta dalle **Blocking-Queue** può rivelarsi considerevole: vengono eliminati l'abbinamento tra le classi esistenti e la chiamata esplicita dei metodi **wait()** e **notifyAll()**, poiché ogni classe comunica soltanto con la sua **BlockingQueue**.

**Esercizio 29** (8) Modificate **ToastOMatic.java** per produrre toast con burro di arachidi e gelatina di frutta utilizzando due linee di produzione separate, una per il burro di arachidi e la seconda per la gelatina. Infine unificate le due linee di produzione.

### **Utilizzo delle pipe per l'I/O tra i task**

Spesso è utile che i task comunicino tra loro utilizzando le funzionalità di I/O. Le librerie di threading supportano le operazioni di I/O "intertask" ricorrendo alle *pipe*, presenti nella libreria di I/O di Java sotto forma delle



classi **PipedWriter**, che esegue un'operazione di scrittura in una pipe, e **PipedReader**, che esegue la lettura dalla stessa pipe. Questi meccanismi ricordano una variante del problema produttore-consamatore, del quale la pipe è una soluzione preconfezionata. In pratica la pipe è una coda bloccante, già presente in versioni di Java precedenti l'introduzione di **BlockingQueue**.

Ecco un semplice esempio di comunicazione tra due task che avviene attraverso una pipe.

```
//: concurrency/PipedIO.java
// Utilizzo delle pipe per le comunicazioni tra i task
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Sender implements Runnable {
    private Random rand = new Random(47);
    private PipedWriter out = new PipedWriter();
    public PipedWriter getPipedWriter() { return out; }
    public void run() {
        try {
            while(true)
                for(char c = 'A'; c <= 'z'; c++) {
                    out.write(c);
                    TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
                }
        } catch(IOException e) {
            print(e + " Sender write exception");
        } catch(InterruptedException e) {
            print(e + " Sender sleep interrupted");
        }
    }
}

class Receiver implements Runnable {
    private PipedReader in;
    public Receiver(Sender sender) throws IOException {

```



```
    in = new PipedReader(sender.getPipedWriter());
}
public void run() {
    try {
        while(true) {
            // Si blocca fino a quando non giungono caratteri:
            printnb("Read: " + (char)in.read() + ", ");
        }
    } catch(IOException e) {
        print(e + " Receiver read exception");
    }
}

public class PipedIO {
    public static void main(String[] args) throws Exception {
        Sender sender = new Sender();
        Receiver receiver = new Receiver(sender);
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(sender);
        exec.execute(receiver);
        TimeUnit.SECONDS.sleep(4);
        exec.shutdownNow();
    }
} /* Output: (65% match)
Read: A, Read: B, Read: C, Read: D, Read: E, Read: F, Read: G,
Read: H, Read: I, Read: J, Read: K, Read: L, Read: M,
java.lang.InterruptedException: sleep interrupted Sender
sleep interrupted
java.io.InterruptedIOException Receiver read exception
*///:-
```

**Sender** e **Receiver** rappresentano i task che devono comunicare. **Sender** crea un **PipedWriter**, che è un oggetto autonomo; in **Receiver**, invece, nella chiamata al costruttore la creazione di **PipedReader** deve essere associata a un **PipedWriter**. **Sender** immette i dati in **Writer**, poi esegue una pausa con **sleep()** per un periodo di tempo determinato casualmente. Tuttavia **Receiver** non



esegue chiamate a `sleep()` o `wait()`, ma quando chiama `read()` per eseguire la lettura la pipe si blocca automaticamente non appena terminano i dati.

Notate che **sender** e **receiver** vengono avviati nel metodo `main()`, dopo che gli oggetti sono stati interamente costruiti. Se non avviate oggetti completamente creati, la pipe potrebbe dare luogo a comportamenti incoerenti in funzione delle piattaforme utilizzate: ricordate che le **BlockingQueue** sono più robuste e più facili da utilizzare.

Una differenza importante tra un **PipedReader** e una normale classe di I/O si nota nella chiamata al metodo `shutdownNow()`: **PipedReader** è interrompibile, mentre se cambiaste, per esempio, la chiamata `in.read()` in `System.in.read()`, il metodo `interrupt()` non riuscirebbe a interrompere la chiamata al metodo `read()`.

**Esercizio 30 (1)** Modificate `PipedIO.java` per utilizzare una **Blocking-Queue** invece di una pipe.

## Deadlock

Ora sapete che un oggetto può avere metodi **synchronized** o altre forme di lock che impediscono ai task di accedere all'oggetto finché il mutex non venga rilasciato. Avete anche visto che i task possono essere bloccati. È quindi possibile che un task rimanga bloccato in attesa di un secondo task, che a sua volta ne aspetta un terzo e così via, fino a quando la catena non conduca finalmente a un task in attesa del primo. Si avrà così un ciclo continuo di task che si attendono a vicenda, in cui tutti sono impossibilitati a muoversi: questa condizione è chiamata *stallo* o *punto morto* (*deadlock*).<sup>17</sup>

Se cercate di eseguire un programma e giungete subito a un punto morto, non avrete difficoltà a rintracciare il bug. Il problema diventa più serio quando il programma sembra funzionare alla perfezione, tuttavia ha la potenzialità di giungere a un punto morto. In tal caso potreste non avere alcuna indicazione della possibilità di deadlock, quindi il difetto è destinato a rimanere latente nel programma fino a quando non si verificherà presso un cliente, di norma in un modo che sarà difficile da riprodurre. Quindi, una parte critica nello sviluppo dei sistemi concorrenti è la prevenzione dei punti morti ottenibile grazie a un'adeguata progettazione.

Il problema della “cena dei filosofi” inventato da Edsger Dijkstra ([http://it.wikipedia.org/wikil/Problema\\_dei\\_filosofi](http://it.wikipedia.org/wikil/Problema_dei_filosofi)) rappresenta la classica dimostrazione della situazione di stallo. La descrizione originale presenta cinque filo-

17. Si può verificare anche un *livelock* (letteralmente, “punto vivo”) quando due task possono cambiare la loro condizione, senza bloccarsi e senza tuttavia realizzare alcun progresso utile.



sofi, ma l'esempio seguente ne permetterà un numero arbitrario: ciascun filosofo dedica una parte del suo tempo a pensare e una parte a nutrirsi. Mentre stanno pensando i filosofi non necessitano di alcuna risorsa comune, ma quando mangiano utilizzano un numero limitato di posate. Nella descrizione originale del problema gli utensili sono due forchette, che ogni filosofo utilizza per prendere gli spaghetti dal piatto; in alcune versioni e nella descrizione che segue si fa riferimento alle bacchette cinesi, ma in ogni caso il problema non cambia: ogni filosofo avrà bisogno di due bacchette per prendere gli spaghetti, in modo da non restare digiuno.

Il problema introduce una difficoltà: poiché i filosofi dispongono di pochissimo denaro non possono permettersi altro che cinque bacchette: in un'ottica più generale il numero di bacchette è pari a quello dei filosofi. Le bacchette sono disposte a lato dei piatti, tra un coperto e l'altro; quando un filosofo vuole mangiare deve prendere la bacchetta di sinistra e quella di destra. Se un altro commensale da qualsiasi lato sta utilizzando una delle due bacchette, il povero filosofo dovrà attendere che entrambe le bacchette siano disponibili.

```
//: concurrency/Chopstick.java
// Bacchette cinesi nella cena dei filosofi.

public class Chopstick {
    private boolean taken = false;
    public synchronized
    void take() throws InterruptedException {
        while(taken)
            wait();
        taken = true;
    }
    public synchronized void drop() {
        taken = false;
        notifyAll();
    }
} //:~
```

Non è possibile che due **Philosopher** possano prendere (**take()**) la stessa bacchetta (**Chopstick**) contemporaneamente. Inoltre, se una **Chopstick** è in mano a un **Philosopher**, un altro dovrà rimanere in attesa (**wait()**) che la **Chopstick** diventi disponibile, ovvero che il filosofo corrente appoggi (**drop()**) la bacchetta sul tavolo.



Quando un task **Philosopher** chiama `take()`, questo **Philosopher** attende finché il flag **taken** non diventi `false`, ossia fino a quando il **Philosopher** che in quel momento ha in mano la **Chopstick** non la rilasci. A quel punto il task imposta il flag **taken** a `true` per indicare il possesso della **Chopstick** da parte di un altro **Philosopher**. Quando questo **Philosopher** ha terminato di utilizzare la **Chopstick** chiama `drop()` per modificare il flag e avvisa, con `notifyAll()`, tutti gli altri **Philosopher** che potenzialmente erano in attesa (`wait()`) della **Chopstick**.

```
//: concurrency/Philosopher.java
// Un filosofo a cena
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Philosopher implements Runnable {
    private Chopstick left;
    private Chopstick right;
    private final int id;
    private final int ponderFactor;
    private Random rand = new Random(47);
    private void pause() throws InterruptedException {
        if(ponderFactor == 0) return;
        TimeUnit.MILLISECONDS.sleep(
            rand.nextInt(ponderFactor * 250));
    }
    public Philosopher(Chopstick left, Chopstick right,
        int ident, int ponder) {
        this.left = left;
        this.right = right;
        id = ident;
        ponderFactor = ponder;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                print(this + " " + "thinking");
                pause();
                if(left.taken)
                    left.drop();
                if(right.taken)
                    right.drop();
                if(left.taken)
                    left.take();
                if(right.taken)
                    right.take();
                print(this + " " + "eating");
                pause();
                if(left.taken)
                    left.drop();
                if(right.taken)
                    right.drop();
                if(left.taken)
                    left.take();
                if(right.taken)
                    right.take();
                print(this + " " + "thinking");
                pause();
            }
        } catch(InterruptedException e) {
            print(this + " " + "interrupted");
        }
    }
}
```



```
// Il filosofo inizia ad avere fame
print(this + " " + "grabbing right");
right.take();
print(this + " " + "grabbing left");
left.take();
print(this + " " + "eating");
pause();
right.drop();
left.drop();
}
} catch(InterruptedException e) {
print(this + " " + "exiting via interrupt");
}
}
public String toString() { return "Philosopher " + id; }
} //:-~
```

In **Philosopher.run()**, ogni **Philosopher** pensa e mangia continuamente. Il metodo **pause()** rimane in pausa (**sleep()**) per un certo periodo di tempo se **ponderFactor** è diverso da zero. In questo modo un **Philosopher** pensa per un periodo di tempo determinato casualmente, quindi prova a prendere (**take()**) le **Chopstick** a destra e a sinistra, mangia per un altro periodo di tempo casuale e così via. Ora è possibile impostare una versione del programma che produrrà una situazione di stallo.

```
//: concurrency/DeadlockingDiningPhilosophers.java
// Dimostra una situazione di stallo potenziale nel programma.
// {Args: 0 5 timeout}
import java.util.concurrent.*;

public class DeadlockingDiningPhilosophers {
    public static void main(String[] args) throws Exception {
        int ponder = 5;
        if(args.length > 0)
            ponder = Integer.parseInt(args[0]);
        int size = 5;
        if(args.length > 1)
            size = Integer.parseInt(args[1]);
```



```

ExecutorService exec = Executors.newCachedThreadPool();
Chopstick[] sticks = new Chopstick[size];
for(int i = 0; i < size; i++)
    sticks[i] = new Chopstick();
for(int i = 0; i < size; i++)
    exec.execute(new Philosopher(
        sticks[i], sticks[(i+1) % size], i, ponder));
if(args.length == 3 && args[2].equals("timeout"))
    TimeUnit.SECONDS.sleep(5);
else {
    System.out.println("Press 'Enter' to quit");
    System.in.read();
}
exec.shutdownNow();
}
} /* (Da eseguire per visualizzare l'output) *///:-

```

Noterete che se i **Philosopher** passano poco tempo pensando, tutti saranno in competizione per le **Chopstick** necessarie per mangiare e la condizione di stallo si attuerà molto più in fretta.

Il primo argomento da riga di comando imposta il fattore **ponder**, che determina il tempo dedicato da ogni **Philosopher** alla riflessione. Se alla cena partecipano molti **Philosopher**, o se essi passano molto tempo a pensare, potrete non arrivare mai al punto morto, sebbene questo rimanga una possibilità reale. Se l'argomento fornito da riga di comando è zero la situazione di stallo sarà raggiunta molto rapidamente.

Notate che gli oggetti **Chopstick** non necessitano di identificativi interni, poiché sono riconoscibili dalla loro posizione nell'array **sticks**. Il costruttore di ciascun **Philosopher** acquisisce un riferimento a un oggetto **Chopstick**, a sinistra e a destra. Ogni **Philosopher**, tranne l'ultimo, viene inizializzato posizionandolo tra la coppia di oggetti **Chopstick** successiva. All'ultimo **Philosopher** viene assegnata la **Chopstick** numero zero, ossia la prima, come **Chopstick** di destra: in questo modo la tavola rotonda si completa, poiché l'ultimo **Philosopher** si siede accanto al primo, ed entrambi condividono la **Chopstick** numero zero. A quel punto tutti i **Philosopher** potranno provare a mangiare, attendendo che il **Philosopher** loro vicino posi la sua **Chopstick**. Questo porterà il programma al punto morto.

Se i **Philosopher** stanno dedicando più tempo alla meditazione che alla tavola, avranno una probabilità di gran lunga inferiore di richiedere le risorse comuni (**Chopstick**); potrete così ritenere che il programma sia esente da



situazioni di stallo, anche se in realtà non è così: questa convinzione deriva dal comportamento che si ottiene impiegando un valore **ponder** diverso da zero o un numero elevato di **Philosopher**. L'esempio è interessante poiché dimostra che un programma può dare la falsa impressione di funzionare correttamente, mentre in realtà potrebbe giungere a un punto morto.

Per risolvere il problema dovete comprendere che lo stallo può avvenire quando si verificano simultaneamente quattro condizioni.

1. Esclusione reciproca. Almeno una risorsa utilizzata dai task non deve essere condivisibile. In questo caso, una **Chopstick** può essere utilizzata soltanto da un **Philosopher** per volta.
2. Almeno un task deve avere una risorsa e attendere di acquisire una risorsa che in quel momento è in possesso di un altro task. In altri termini, affinché il punto morto si avveri, un **Philosopher** deve detenere una **Chopstick** e attendere di acquisire l'altra.
3. Una risorsa non può essere tolta a un task in anticipo. I task liberano le risorse come normale evento della cena: i **Philosopher** sono educati e non strappano di mano la **Chopstick** a un altro **Philosopher**.
4. Può verificarsi una situazione di "attesa circolare": un task aspetta la risorsa in possesso di un altro task, che a sua volta attende quella detenuta da un altro e così via, finché uno dei task non stia attendendo la risorsa posseduta dal primo task, generando così il blocco totale. In **Deadlocking-DiningPhilosophers.java** si riscontra questo fenomeno di attesa circolare, poiché ogni **Philosopher** cerca in primo luogo di ottenere la **Chopstick** a destra e solo in seguito quella di sinistra.

Poiché la condizione essenziale affinché si verifichi un punto morto è che tutte queste circostanze vengano soddisfatte, è sufficiente eliminarne una per impedire che avvenga lo stallo. In questo programma, il modo più semplice per rendere impossibile il verificarsi di un punto morto consiste nell'imperdere la quarta condizione. Questa si produce poiché ogni **Philosopher** cerca di prendere le **Chopstick** in una sequenza specifica: prima a destra, poi a sinistra. A causa di questa particolare progressione è possibile che ciascun filosofo possieda la **Chopstick** di destra e sia in attesa di ottenere quella di sinistra, dando origine al fenomeno di attesa circolare.

Tuttavia, se l'inizializzazione dell'ultimo **Philosopher** prevedesse come primo obiettivo l'ottenimento della **Chopstick** di sinistra, non sarebbe in alcun modo possibile che il filosofo impedisca al **Philosopher** a destra di prelevare la propria bacchetta: così facendo si eviterà l'attesa circolare. Naturalmente questa è soltanto una delle possibili soluzioni del problema, che potrebbe essere risolto anche impedendo una qualsiasi delle altre circostanze: per mag-



giori dettagli consultate un manuale dedicato al threading avanzato o fate riferimento alla bibliografia al termine del capitolo.

```
//: concurrency/FixedDiningPhilosophers.java
// La cena dei filosofi senza deadlock.
// {Args: 5 5 timeout}
import java.util.concurrent.*;

public class FixedDiningPhilosophers {
    public static void main(String[] args) throws Exception {
        int ponder = 5;
        if(args.length > 0)
            ponder = Integer.parseInt(args[0]);
        int size = 5;
        if(args.length > 1)
            size = Integer.parseInt(args[1]);
        ExecutorService exec = Executors.newCachedThreadPool();
        Chopstick[] sticks = new Chopstick[size];
        for(int i = 0; i < size; i++)
            sticks[i] = new Chopstick();
        for(int i = 0; i < size; i++)
            if(i < (size-1))
                exec.execute(new Philosopher(
                    sticks[i], sticks[i+1], i, ponder));
            else
                exec.execute(new Philosopher(
                    sticks[0], sticks[i], i, ponder));
        if(args.length == 3 && args[2].equals("timeout"))
            TimeUnit.SECONDS.sleep(5);
        else {
            System.out.println("Press 'Enter' to quit");
            System.in.read();
        }
        exec.shutdownNow();
    }
} /* (Da eseguire per visualizzare l'output) *///:~
```



Facendo in modo che l'ultimo **Philosopher** prenda e posa la **Chopstick** alla sua sinistra prima di quella a destra, rimuoverete il punto morto e il programma funzionerà uniformemente.

Il linguaggio Java non offre supporto alla prevenzione delle situazioni di stallo: spetta a voi evitarle grazie a una progettazione meticolosa. Certo, queste parole non saranno di conforto a chi stesse cercando di far funzionare un programma arrivato a un punto morto.

**Esercizio 31** (8) Modificate **DeadlockingDiningPhilosophers.java** in modo che quando un filosofo ha utilizzato le sue bacchette, le sistemi su un piattino davanti a sé. Quando un filosofo desidera mangiare, prende le due bacchette disponibili dal piattino. Questa modifica è sufficiente per eliminare la possibilità che avvenga un punto morto? È possibile reintrodurre la situazione di stallo semplicemente riducendo il numero di bacchette disponibili?

## Nuovi componenti di libreria

La libreria **java.util.concurrent** in Java SE5 ha introdotto molte nuove classi destinate a risolvere i problemi di concorrenza: imparare a utilizzarle potrà esservi di aiuto nella realizzazione di programmi concorrenti più semplici e robusti.

I prossimi paragrafi mostrano un insieme rappresentativo dei vari componenti; tuttavia alcuni di essi, quelli che avrete probabilmente meno occasione di trovare e utilizzare nel codice, non saranno esaminati.

Poiché questi componenti risolvono problemi di vario tipo, non è stato possibile classificarli in modo organico, pertanto inizierete con gli esempi più semplici e continuerete con altri via via più complessi.

### ***CountDownLatch***

Questa classe è utilizzata per sincronizzare uno o più task costringendoli ad attendere il completamento di un insieme di operazioni condotte da altri task.

Inizierete assegnando un valore iniziale al contatore di un oggetto **CountDownLatch**, e ogni task che esegue una chiamata ad **await()** su quell'oggetto rimarrà bloccato finché il contatore non arriverà a zero. Altri task possono chiamare **countDown()** sull'oggetto per diminuire il contatore, presumibilmente quando terminano la loro attività. Una classe **CountDownLatch** è destinata a essere utilizzata una sola volta, poiché il contatore non può essere azzerato: se avete bisogno di questa funzionalità, potete servirvi di **CyclicBarrier**.



I task che chiamano `countDown()` non sono bloccati nel momento in cui eseguono la chiamata; soltanto la chiamata `await()` rimane bloccata fino a quando il contatore non giunge a zero.

Un utilizzo tipico di `CountDownLatch` consiste nel suddividere il problema di programmazione in  $n$  task indipendenti e risolvibili e creare un `CountDownLatch` con il valore  $n$ . Quando ogni task è terminato chiama `countDown()`, come se azionasse una “serratura a tempo”. I task che aspettano la soluzione del problema chiamano `await()` per trattenersi fino al suo completamento. Ecco una struttura di esempio schematica che dimostra questa tecnica.<sup>18</sup>

```
//: concurrency/CountDownLatchDemo.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Esegue alcune porzioni di un task:
class TaskPortion implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private static Random rand = new Random(47);
    private final CountDownLatch latch;
    TaskPortion(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            doWork();
            latch.countDown();
        } catch(InterruptedException ex) {
            // Un modo accettabile per uscire
        }
    }
    public void doWork() throws InterruptedException {
        TimeUnit.MILLISECONDS.sleep(rand.nextInt(2000));
    }
}
```

18. `CountDownLatch` è uno strumento di sincronizzazione che permette a uno o più thread di rimanere in attesa del completamento di operazioni eseguite da altri thread, una condizione definita in gergo *on the latch*, che potrebbe tradursi come “chiavistello apribile”.



```
    print(this + " completed");
}
public String toString() {
    return String.format("%1$-3d ", id);
}
}

// Rimane in attesa su CountDownLatch:
class WaitingTask implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final CountDownLatch latch;
    WaitingTask(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            latch.await();
            print("Latch barrier passed for " + this);
        } catch(InterruptedException ex) {
            print(this + " interrupted");
        }
    }
    public String toString() {
        return String.format("WaitingTask %1$-3d ", id);
    }
}

public class CountDownLatchDemo {
    static final int SIZE = 100;
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        // Tutti i task devono condividere un unico oggetto
        // CountDownLatch:
        CountDownLatch latch = new CountDownLatch(SIZE);
        for(int i = 0; i < 10; i++)
            exec.execute(new WaitingTask(latch));
    }
}
```

```

for(int i = 0; i < SIZE; i++)
    exec.execute(new TaskPortion(latch));
print("Launched all tasks");
exec.shutdown(); // Termina quando tutti i task sono stati
                 // completati
}
} /* (Da eseguire per visualizzare l'output) *///:~

```

**TaskPortion** rimane in attesa per un periodo di tempo determinato casualmente al fine di simulare l'ultimazione di una parte del task, e **WaitingTask** indica la parte del sistema che deve attendere il completamento della parte iniziale del problema. Tutti i task funzionano con lo stesso **CountDownLatch**, definito in **main()**.

**Esercizio 32 (7)** Utilizzate un **CountDownLatch** per risolvere il problema di correlazione dei risultati delle **Entrance** in **OrnamentalGarden.java**. Eliminate il codice inutile dalla nuova versione dell'esempio.

### Funzionalità di libreria per la sicurezza dei thread

Notate che **TaskPortion** contiene un oggetto **Random static**, per indicare che diversi task potrebbero chiamare **Random.nextInt()** nello stesso momento. Si tratta di un approccio sicuro?

Per rispondere alla domanda occorre tenere presente che, in questo caso, un eventuale problema potrebbe essere risolto assegnando a **TaskPortion** il suo oggetto **Random**, vale a dire rimuovendo l'indicatore **static**. Tuttavia la domanda è sempre valida, in generale, per i metodi della libreria standard di Java: quali thread sono sicuri e quali non lo sono?

Purtroppo la documentazione di JDK non è molto chiara in proposito. Viene detto che **Random.nextInt()** è un thread sicuro, purtroppo però dovete renderne conto caso per caso, eseguendo ricerche su Internet o analizzando il codice della libreria Java: una situazione decisamente non ottimale per un linguaggio di programmazione che, almeno nella teoria, dovrebbe supportare la concorrenza.

### CyclicBarrier

La classe **CyclicBarrier** viene utilizzata nelle situazioni in cui occorra creare un gruppo di task per eseguire attività parallele, che poi rimangono in attesa del loro completamento prima di passare al punto seguente: un approccio simile a quello di **join()**, all'apparenza. **CyclicBarrier** porta tutti i task paral-



leli sulla “barriera di partenza”, in modo che possano procedere all'unisono. Questo comportamento è molto simile a quello di **CountDownLatch**, tranne per il fatto che un **CountDownLatch** è un evento unico mentre **CyclicBarrier** può essere riutilizzata più volte.

Gli emulatori hanno sempre affascinato l'autore fin dall'inizio della sua esperienza con i computer, e la concorrenza è un fattore chiave per realizzare le simulazioni. Il primo programma che l'autore ricorda di avere scritto era una simulazione, un gioco scritto in BASIC chiamato HOSRAC.BAS. Ecco la versione orientata agli oggetti di questo programma, con il supporto per i thread, che utilizza una **CyclicBarrier**.

```
//: concurrency/HorseRace.java
// Utilizzo di CyclicBarrier.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Horse implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private int strides = 0;
    private static Random rand = new Random(47);
    private static CyclicBarrier barrier;
    public Horse(CyclicBarrier b) { barrier = b; }
    public synchronized int getStrides() { return strides; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    strides += rand.nextInt(3); // Produce 0, 1 o 2
                }
                barrier.await();
            }
        } catch(InterruptedException e) {
            // Un modo legittimo per uscire
        } catch(BrokenBarrierException e) {
            // Di questo sarebbe interessante sapere qualcosa di piu'
        }
    }
}
```



```
        throw new RuntimeException(e);
    }
}

public String toString() { return "Horse " + id + " "; }
public String tracks() {
    StringBuilder s = new StringBuilder();
    for(int i = 0; i < getStrides(); i++)
        s.append("*");
    s.append(id);
    return s.toString();
}
}

public class HorseRace {
    static final int FINISH_LINE = 75;
    private List<Horse> horses = new ArrayList<Horse>();
    private ExecutorService exec =
        Executors.newCachedThreadPool();
    private CyclicBarrier barrier;
    public HorseRace(int nHorses, final int pause) {
        barrier = new CyclicBarrier(nHorses, new Runnable() {
            public void run() {
                StringBuilder s = new StringBuilder();
                for(int i = 0; i < FINISH_LINE; i++)
                    s.append("-"); // Il divisorio del tracciato di gara
                print(s);
                for(Horse horse : horses)
                    print(horse.tracks());
                for(Horse horse : horses)
                    if(horse.getStrides() >= FINISH_LINE) {
                        print(horse + "won!");
                        exec.shutdownNow();
                        return;
                    }
            }
        try {
            TimeUnit.MILLISECONDS.sleep(pause);
        }
```



```
        } catch(InterruptedException e) {
            print("barrier-action sleep interrupted");
        }
    }
}:
for(int i = 0; i < nHorses; i++) {
    Horse horse = new Horse(barrier);
    horses.add(horse);
    exec.execute(horse);
}
}
public static void main(String[] args) {
    int nHorses = 7;
    int pause = 200;
    if(args.length > 0) { // Argomento opzionale
        int n = new Integer(args[0]);
        nHorses = n > 0 ? n : nHorses;
    }
    if(args.length > 1) { // Argomento opzionale
        int p = new Integer(args[1]);
        pause = p > -1 ? p : pause;
    }
    new HorseRace(nHorses, pause);
}
/* (Da eseguire per visualizzare l'output) *///:~
```

A una classe **CyclicBarrier** può essere assegnata un’“azione barriera”, ovvero un **Runnable** che viene eseguito automaticamente quando il contatore arriva a zero: questa è un’altra differenza tra **CyclicBarrier** e **CountDownLatch**. In questo caso specifico l’azione barriera viene creata come classe anonima, poi passata al costruttore di **CyclicBarrier**.

L’autore ha cercato di fare in modo che ogni cavallo visualizzasse le proprie informazioni, ma l’ordine di visualizzazione dipendeva dal gestore dei task. **CyclicBarrier** fa sì che ogni cavallo possa spostarsi in avanti, quindi attende che tutti gli altri cavalli procedano. Quando tutti i cavalli si sono mossi, **CyclicBarrier** chiama automaticamente il task di azione barriera **Runnable** per visualizzare i cavalli nell’ordine, con il divisorio del tracciato di gara.



Una volta che tutti i task hanno superato la barriera, il programma è automaticamente pronto per il prossimo giro.

Per rendere ben visibile il funzionamento del programma, dovete ridimensionare la finestra di console, in modo che sia visualizzata soltanto una serie di cavalli, da 0 a 6.

### ***DelayQueue***

La classe **DelayQueue** implementa l'interfaccia **BlockingQueue** ed estende l'interfaccia **Delayed**. Un oggetto può essere prelevato da **DelayQueue\_CORSIVO** solo quando il suo ritardo è scaduto, e la coda è ordinata in modo che l'oggetto iniziale corrisponda al ritardo scaduto da più tempo. Se il ritardo non è scaduto per nessun elemento, non verrà selezionato alcun elemento di testa e il metodo **poll()** restituirà **null**; per questa ragione non è possibile inserire elementi **null** nella coda.

Nell'esempio seguente gli oggetti **Delayed** sono essi stessi task, in quanto **DelayedTask** implementa **Delayed**; **DelayedTaskConsumer** acquisisce ed esegue dalla coda il task più "urgente", ovvero quello scaduto da più tempo. Notate che **DelayQueue** è una variazione di una coda con priorità.

```
//: concurrency/DelayQueueDemo.java
import java.util.concurrent.*;
import java.util.*;
import static java.util.concurrent.TimeUnit.*;
import static net.mindview.util.Print.*;

class DelayedTask implements Runnable, Delayed {
    private static int counter = 0;
    private final int id = counter++;
    private final int delta;
    private final long trigger;
    protected static List<DelayedTask> sequence =
        new ArrayList<DelayedTask>();
    public DelayedTask(int delayInMilliseconds) {
        delta = delayInMilliseconds;
        trigger = System.nanoTime() +
            NANOSECONDS.convert(delta, MILLISECONDS);
        sequence.add(this);
    }
}
```



```
public long getDelay(TimeUnit unit) {
    return unit.convert(
        trigger - System.nanoTime(), NANOSECONDS);
}
public int compareTo(Delayed arg) {
    DelayedTask that = (DelayedTask)arg;
    if(trigger < that.trigger) return -1;
    if(trigger > that.trigger) return 1;
    return 0;
}
public void run() { printnb(this + " "); }
public String toString() {
    return String.format("[%1$-4d]", delta) +
        " Task " + id;
}
public String summary() {
    return "(" + id + ":" + delta + ")";
}
public static class EndSentinel extends DelayedTask {
    private ExecutorService exec;
    public EndSentinel(int delay, ExecutorService e) {
        super(delay);
        exec = e;
    }
    public void run() {
        for(DelayedTask pt : sequence) {
            printnb(pt.summary() + " ");
        }
        print();
        print(this + " Calling shutdownNow()");
        exec.shutdownNow();
    }
}
}

class DelayedTaskConsumer implements Runnable {
    private DelayQueue<DelayedTask> q;
```



```
public DelayedTaskConsumer(DelayQueue<DelayedTask> q) {
    this.q = q;
}
public void run() {
    try {
        while(!Thread.interrupted())
            q.take().run(); // Esegue il task con il thread
                           // corrente
    } catch(InterruptedException e) {
        // Un modo accettabile per uscire
    }
    print("Finished DelayedTaskConsumer");
}
}

public class DelayQueueDemo {
    public static void main(String[] args) {
        Random rand = new Random(47);
        ExecutorService exec = Executors.newCachedThreadPool();
        DelayQueue<DelayedTask> queue =
            new DelayQueue<DelayedTask>();
        // Popola con task che hanno ritardi casuali:
        for(int i = 0; i < 20; i++)
            queue.put(new DelayedTask(rand.nextInt(5000)));
        // Imposta il punto di stop
        queue.add(new DelayedTask.EndSentinel(5000, exec));
        exec.execute(new DelayedTaskConsumer(queue));
    }
} /* Output:
[128 ] Task 11 [200 ] Task 7 [429 ] Task 5 [520 ] Task 18
[555 ] Task 1 [961 ] Task 4 [998 ] Task 16 [1207] Task 9
[1693] Task 2 [1809] Task 14 [1861] Task 3 [2278] Task 15
[3288] Task 10 [3551] Task 12 [4258] Task 0 [4258] Task 19
[4522] Task 8 [4589] Task 13 [4861] Task 17 [4868] Task 6
(0:4258) (1:555) (2:1693) (3:1861) (4:961) (5:429) (6:4868)
(7:200) (8:4522) (9:1207) (10:3288) (11:128) (12:3551)
```



```
(13:4589) (14:1809) (15:2278) (16:998) (17:4861) (18:520)
(19:4258) (20:5000)
[5000] Task 20 Calling shutdownNow()
Finished DelayedTaskConsumer
*///:~
```

**DelayedTask** contiene un oggetto **List<DelayedTask>** chiamato **sequence** che conserva l'ordine in cui i task sono stati creati.

L'interfaccia **Delayed** ha il metodo **getDelay()**, che indica il tempo mancante alla scadenza del ritardo (*delay*) o il tempo trascorso dalla scadenza del ritardo; questo metodo accetta argomenti di tipo **TimeUnit**, quindi dovrete servirvi di quest'ultima classe, peraltro molto pratica poiché consente di convertire facilmente le unità di misura senza dover effettuare calcoli. Per esempio, sebbene il valore di **delta** sia memorizzato in millisecondi, il metodo **System.nanoTime()** di Java SE5 restituisce il valore espresso in nanosecondi. Potete convertire il valore **delta** indicando l'unità di misura originale e l'unità desiderata, come nel seguente esempio:

```
NANOSECONDS.convert(delta, MILLISECONDS);
```

In **getDelay()** l'unità desiderata è passata come parametro **unit**, utilizzato per convertire la differenza tra il tempo di attivazione (*trigger time*) e le unità di tempo richieste dal chiamante, senza dover sapere a quanto corrispondano tali unità: questo è un semplice esempio del design pattern *Strategy*, di cui parte dell'algoritmo viene passata come argomento.

Per l'ordinamento l'interfaccia **Delayed** eredita anche l'interfaccia **Comparable**, quindi è necessario implementare il metodo **compareTo()** in modo che produca un confronto ragionevole. I metodi **toString()** e **summary()** eseguono la formattazione dell'output; la classe nidificata **EndSentinel** fornisce un metodo per chiudere tutto inserendola come ultimo elemento nella coda.

Tenetevi presente che, poiché **DelayedTaskConsumer** è di per sé un task, possiede il proprio **Thread** che può essere utilizzato per eseguire ogni task che esce dalla coda. Dal momento che i task vengono eseguiti nell'ordine di priorità della coda, in questo esempio non è necessario avviare thread separati per eseguire **DelayedTask**.

L'output evidenzia che l'ordine di creazione dei task non ha effetto sull'ordine di esecuzione: i task vengono eseguiti nell'ordine di ritardo, come previsto.



## PriorityBlockingQueue

Si tratta essenzialmente di una coda con priorità che dispone di funzionalità di recupero bloccanti. In questo esempio gli oggetti presenti nella coda (con priorità) sono task che escono dalla coda in ordine di priorità. A un **PrioritizedTask** viene assegnato un numero (di priorità) che rappresenta questo ordine.

```
//: concurrency/PriorityBlockingQueueDemo.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class PrioritizedTask implements
Runnable, Comparable<PrioritizedTask> {
    private Random rand = new Random(47);
    private static int counter = 0;
    private final int id = counter++;
    private final int priority;
    protected static List<PrioritizedTask> sequence =
        new ArrayList<PrioritizedTask>();
    public PrioritizedTask(int priority) {
        this.priority = priority;
        sequence.add(this);
    }
    public int compareTo(PrioritizedTask arg) {
        return priority < arg.priority ? 1 :
            (priority > arg.priority ? -1 : 0);
    }
    public void run() {
        try {
            TimeUnit.MILLISECONDS.sleep(rand.nextInt(250));
        } catch(InterruptedException e) {
            // Un modo accettabile per uscire
        }
        print(this);
    }
    public String toString() {
```



```
        return String.format("[%1$-3d]", priority) +
               " Task " + id;
    }
    public String summary() {
        return "(" + id + ":" + priority + ")";
    }
    public static class EndSentinel extends PrioritizedTask {
        private ExecutorService exec;
        public EndSentinel(ExecutorService e) {
            super(-1); // Priorita' piu' bassa in questo programma
            exec = e;
        }
        public void run() {
            int count = 0;
            for(PrioritizedTask pt : sequence) {
                printnb(pt.summary());
                if(++count % 5 == 0)
                    print();
            }
            print();
            print(this + " Calling shutdownNow()");
            exec.shutdownNow();
        }
    }
}

class PrioritizedTaskProducer implements Runnable {
    private Random rand = new Random(47);
    private Queue<Runnable> queue;
    private ExecutorService exec;
    public PrioritizedTaskProducer(
        Queue<Runnable> q, ExecutorService e) {
        queue = q;
        exec = e; // utilizzato per EndSentinel
    }
    public void run() {
        // Coda senza limiti; nessun blocco.
```



```
// Popolata rapidamente con priorita' casuali:  
for(int i = 0; i < 20; i++) {  
    queue.add(new PrioritizedTask(rand.nextInt(10)));  
    Thread.yield();  
}  
// Inserisce i job ad alta priorita':  
try {  
    for(int i = 0; i < 10; i++) {  
        TimeUnit.MILLISECONDS.sleep(250);  
        queue.add(new PrioritizedTask(10));  
    }  
    // Aggiunge i task, prima quelli a bassa priorita':  
    for(int i = 0; i < 10; i++)  
        queue.add(new PrioritizedTask(i));  
    // Una sentinella per interrompere tutti i task:  
    queue.add(new PrioritizedTask.EndSentinel(exec));  
} catch(InterruptedException e) {  
    // Un modo accettabile per uscire  
}  
print("Finished PrioritizedTaskProducer");  
}  
}  
  
class PrioritizedTaskConsumer implements Runnable {  
    private PriorityBlockingQueue<Runnable> q;  
    public PrioritizedTaskConsumer(  
        PriorityBlockingQueue<Runnable> q) {  
        this.q = q;  
    }  
    public void run() {  
        try {  
            while(!Thread.interrupted())  
                // Utilizza il thread corrente per eseguire il task:  
                q.take().run();  
        } catch(InterruptedException e) {  
            // Un modo accettabile per uscire  
        }  
    }  
}
```



```
    print("Finished PrioritizedTaskConsumer");
}
}

public class PriorityBlockingQueueDemo {
    public static void main(String[] args) throws Exception {
        Random rand = new Random(47);
        ExecutorService exec = Executors.newCachedThreadPool();
        PriorityBlockingQueue<Runnable> queue =
            new PriorityBlockingQueue<Runnable>();
        exec.execute(new PrioritizedTaskProducer(queue, exec));
        exec.execute(new PrioritizedTaskConsumer(queue));
    }
} /* (Da eseguire per visualizzare l'output) */:-~
```

Come nell'esempio **DelayQueueDemo.java** precedente, la sequenza di creazione degli oggetti **PrioritizedTask** è registrata nella **List sequence**, affinché sia confrontata con l'ordine di esecuzione effettivo. Il metodo **run()** attende per un breve periodo di durata casuale, poi visualizza le informazioni sull'oggetto; **EndSentinel**, come nell'esempio precedente, è l'ultimo elemento inserito nella coda e si occupa di chiudere tutto.

I task **PrioritizedTaskProducer** e **PrioritizedTaskConsumer** sono collegati l'uno all'altro mediante una **PriorityBlockingQueue**. Notate che non è richiesta alcuna sincronizzazione esplicita poiché la natura bloccante della coda fornisce la sincronizzazione necessaria: non dovete preoccuparvi della presenza di elementi nella coda durante la lettura, perché la coda stessa bloccherà il lettore qualora non vi siano elementi.

### ***Un controller di serra con ScheduledExecutor***

Nel Volume I, Capitolo 10 si è utilizzato come esempio un sistema di controllo applicato a un'ipotetica serra, che accende e spegne le varie attrezzature e le impatta. Questo progetto può essere visto come un tipo di problema di concorrenza, in cui ogni evento desiderato per la serra è un task eseguito per un tempo predefinito.

**ScheduledThreadPoolExecutor** fornisce le funzionalità necessarie per risolvere il problema. Potete utilizzare il metodo **schedule()** per eseguire un task una sola volta, o **scheduleAtFixedRate()** per ripeterlo a intervalli specificati, in modo da impostare gli oggetti di tipo **Runnable** da eseguire in futuro. Confrontando l'esempio seguente con il meccanismo utilizzato nel Capitolo 10



precedentemente citato, noterete quanto sia più semplice l'impiego di uno strumento predefinito come **ScheduledThreadPoolExecutor**.

```
//: concurrency/GreenhouseScheduler.java
// Reimplementazione di innerclasses/GreenhouseController.java
// per l'uso di ScheduledThreadPoolExecutor.
// {Args: 5000}
import java.util.concurrent.*;
import java.util.*;

public class GreenhouseScheduler {
    private volatile boolean light = false;
    private volatile boolean water = false;
    private String thermostat = "Day";
    public synchronized String getThermostat() {
        return thermostat;
    }
    public synchronized void setThermostat(String value) {
        thermostat = value;
    }
    ScheduledThreadPoolExecutor scheduler =
        new ScheduledThreadPoolExecutor(10);
    public void schedule(Runnable event, long delay) {
        scheduler.schedule(event,delay,TimeUnit.MILLISECONDS);
    }
    public void
    repeat(Runnable event, long initialDelay, long period) {
        scheduler.scheduleAtFixedRate(
            event, initialDelay, period, TimeUnit.MILLISECONDS);
    }
    class LightOn implements Runnable {
        public void run() {
            // Inserire qui il codice di controllo hardware
            // per accendere le luci.
            System.out.println("Turning on lights");
            light = true;
        }
    }
}
```



```
class LightOff implements Runnable {
    public void run() {
        // Inserire qui il codice di controllo hardware
        // per spegnere le luci.
        System.out.println("Turning off lights");
        light = false;
    }
}
class WaterOn implements Runnable {
    public void run() {
        // Inserire il codice di controllo hardware.
        System.out.println("Turning greenhouse water on");
        water = true;
    }
}
class WaterOff implements Runnable {
    public void run() {
        // Inserire il codice di controllo hardware.
        System.out.println("Turning greenhouse water off");
        water = false;
    }
}
class ThermostatNight implements Runnable {
    public void run() {
        // Inserire il codice di controllo hardware.
        System.out.println("Thermostat to night setting");
        setThermostat("Night");
    }
}
class ThermostatDay implements Runnable {
    public void run() {
        // Inserire il codice di controllo hardware.
        System.out.println("Thermostat to day setting");
        setThermostat("Day");
    }
}
```



```
class Bell implements Runnable {  
    public void run() { System.out.println("Bing!"); }  
}  
// La precedente classe "Restart" non e' necessaria:  
  
class Terminate implements Runnable {  
    public void run() {  
        System.out.println("Terminating");  
        scheduler.shutdownNow();  
        // E' necessario un task separato per questa operazione,  
        // poiche' il programma di pianificazione e' stato chiuso:  
  
        new Thread() {  
            public void run() {  
                for(DataPoint d : data)  
                    System.out.println(d);  
            }  
            }.start();  
    }  
}  
// Nuova funzionalita': raccolta dei dati  
static class DataPoint {  
    final Calendar time;  
    final float temperature;  
    final float humidity;  
    public DataPoint(Calendar d, float temp, float hum) {  
        time = d;  
        temperature = temp;  
        humidity = hum;  
    }  
    public String toString() {  
        return time.getTime() +  
            String.format(  
                " temperature: %1$.1f humidity: %2$.2f",  
                temperature, humidity);  
    }  
}
```



```
private Calendar lastTime = Calendar.getInstance();
{ // Imposta la data alla mezz'ora
    lastTime.set(Calendar.MINUTE, 30);
    lastTime.set(Calendar.SECOND, 00);
}
private float lastTemp = 65.0f;
private int tempDirection = +1;
private float lastHumidity = 50.0f;
private int humidityDirection = +1;
private Random rand = new Random(47);
List<DataPoint> data = Collections.synchronizedList(
    new ArrayList<DataPoint>());
class CollectData implements Runnable {
    public void run() {
        System.out.println("Collecting data");
        synchronized(GreenhouseScheduler.this) {
            // Suppone che l'intervallo sia piu' lungo di quanto
            // non sia effettivamente:
            lastTime.set(Calendar.MINUTE,
                lastTime.get(Calendar.MINUTE) + 30);
            // Una possibilita' su 5 di invertire la direzione:
            if(rand.nextInt(5) == 4)
                tempDirection = -tempDirection;
            // Archivia il valore precedente:
            lastTemp = lastTemp +
                tempDirection * (1.0f + rand.nextFloat());
            if(rand.nextInt(5) == 4)
                humidityDirection = -humidityDirection;
            lastHumidity = lastHumidity +
                humidityDirection * rand.nextFloat();
            // Calendar deve essere clonato, altrimenti tutti i
            // DataPoint mantengono riferimenti allo stesso
            // lastTime.
            // Per un oggetto di base come Calendar,
            // clone() e' OK.
```



```
        data.add(new DataPoint((Calendar)lastTime.clone(),
                               lastTemp, lastHumidity));
    }
}
}

public static void main(String[] args) {
    GreenhouseScheduler gh = new GreenhouseScheduler();
    gh.schedule(gh.new Terminate(), 5000);
    gh.repeat(gh.new Bell(), 0, 1000);
    gh.repeat(gh.new ThermostatNight(), 0, 2000);
    gh.repeat(gh.new LightOn(), 0, 200);
    gh.repeat(gh.new LightOff(), 0, 400);
    gh.repeat(gh.new WaterOn(), 0, 600);
    gh.repeat(gh.new WaterOff(), 0, 800);
    gh.repeat(gh.new ThermostatDay(), 0, 1400);
    gh.repeat(gh.new CollectData(), 500, 500);
}
} /* (Da eseguire per visualizzare l'output) */://:~
```

Questa versione riorganizza il codice e aggiunge una nuova funzionalità per la raccolta dei dati di umidità e temperatura nella serra. Un **DataPoint** mantiene e visualizza una singola porzione di dati, mentre **CollectData** è il task pianificato che genera i dati simulati e li aggiunge a **List<DataPoint>** in **Greenhouse** ogni volta che viene eseguito **CollectData**.

Osservate l'utilizzo delle parole chiave **volatile** e **synchronized**, che impediscono ai task di interferire a vicenda. Alla creazione della **List** viene utilizzato il metodo **synchronizedList()** di **java.util.Collections**, per rendere **synchronized** tutti i metodi della **List** contenente i **DataPoint**.

**Esercizio 33 (7)** Modificate **GreenhouseScheduler.java** in modo che utilizzi una **DelayQueue** invece di uno **ScheduledExecutor**.

## Semafori

Un normale lock, che deriva da **concurrent.locks** o dalla funzionalità nativa **synchronized**, consente a un solo task per volta di accedere a una risorsa; di contro, un *semaforo contatore* (*counting semaphore*) permette a *n* task di accedere contemporaneamente alla risorsa. Potete anche considerare un semaforo come una sorta di “distributore di permessi” per l’utilizzo di una risorsa, benché in realtà non venga utilizzato alcun “oggetto di permesso”.



Come esempio, considerate il concetto di *pool (object pool)* che gestisce un numero limitato di oggetti, “estraendoli” per l’utilizzo e “reintroduceendoli” nel pool quando l’utente ha terminato di servirsene. Questa funzionalità può essere incapsulata in una classe generica.

```
//: concurrency/Pool.java
// Utilizzo di Semaphore in un Pool, per limitare il numero
// di task utilizzabili da una risorsa.
import java.util.concurrent.*;
import java.util.*;

public class Pool<T> {
    private int size;
    private List<T> items = new ArrayList<T>();
    private volatile boolean[] checkedOut;
    private Semaphore available;
    public Pool(Class<T> classObject, int size) {
        this.size = size;
        checkedOut = new boolean[size];
        available = new Semaphore(size, true);
        // Carica il pool con gli oggetti estraibili.
        for(int i = 0; i < size; ++i)
            try {
                // Presuppone un costruttore predefinito:
                items.add(classObject.newInstance());
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
    }
    public T checkOut() throws InterruptedException {
        available.acquire();
        return getItem();
    }
    public void checkIn(T x) {
        if(releaseItem(x))
            available.release();
    }
}
```



```

private synchronized T getItem() {
    for(int i = 0; i < size; ++i)
        if(!checkedOut[i]) {
            checkedOut[i] = true;
            return items.get(i);
        }
    return null; // Semaphore impedisce di giungere a questo
                 // punto
}
private synchronized boolean releaseItem(T item) {
    int index = items.indexOf(item);
    if(index == -1) return false; // Non in elenco
    if(checkedOut[index]) {
        checkedOut[index] = false;
        return true;
    }
    return false; // Non estratto
}
} ///:~

```

In questa forma semplificata il costruttore utilizza `newInstance()` per popolare il pool con gli oggetti. Se vi occorre un nuovo oggetto chiamate `checkOut()` e quando avete terminato lo passate a `checkIn()`.

L'array `boolean checkedOut` tiene traccia degli oggetti estratti e viene gestito tramite i metodi `getItem()` e `releaseItem()`. A loro volta, questi metodi sono custoditi dall'oggetto `available`, di tipo `Semaphore`, così che, in `checkOut()`, `available` blocca la prosecuzione della chiamata se non vi sono più permessi di semaforo disponibili, il che indica l'esaurimento degli oggetti in pool. Se l'oggetto che viene inserito è valido, il metodo `checkIn()` restituisce un permesso al semaforo. Per creare un esempio potete servirvi della classe `Fat`, un tipo di oggetto la cui creazione è considerata onerosa poiché l'esecuzione del suo costruttore richiede un certo tempo.

```

//: concurrency/Fat.java
// Oggetti la cui creazione e' onerosa.

public class Fat {
    private volatile double d; // Impedisce l'ottimizzazione

```



```
private static int counter = 0;
private final int id = counter++;
public Fat() {
    // Operazione onerosa e non interrompibile:
    for(int i = 1; i < 10000; i++) {
        d += (Math.PI + Math.E) / (double)i;
    }
}
public void operation() { System.out.println(this); }
public String toString() { return "Fat id: " + id; }
} //:-~
```

Questi oggetti saranno raggruppati per limitare l'onerosità del costruttore. Potete testare la classe **Pool** creando un task che estrae gli oggetti **Fat**, li mantiene per un certo tempo, poi li inserisce nuovamente.

```
//: concurrency/SemaphoreDemo.java
// Come testare la classe Pool
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Un task per estrarre una risorsa dal pool:
class CheckoutTask<T> implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private Pool<T> pool;
    public CheckoutTask(Pool<T> pool) {
        this.pool = pool;
    }
    public void run() {
        try {
            T item = pool.checkOut();
            print(this + " checked out " + item);
            TimeUnit.SECONDS.sleep(1);
            print(this + " checked in " + item);
            pool.checkIn(item);
        }
```



```
        } catch(InterruptedException e) {
            // Un modo accettabile per uscire
        }
    }
    public String toString() {
        return "CheckoutTask " + id + " ";
    }
}

public class SemaphoreDemo {
    final static int SIZE = 25;
    public static void main(String[] args) throws Exception {
        final Pool<Fat> pool =
            new Pool<Fat>(Fat.class, SIZE);
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < SIZE; i++)
            exec.execute(new CheckoutTask<Fat>(pool));
        print("All CheckoutTasks created");
        List<Fat> list = new ArrayList<Fat>();
        for(int i = 0; i < SIZE; i++) {
            Fat f = pool.checkOut();
            printnb(i + ": main() thread checked out ");
            f.operation();
            list.add(f);
        }
        Future<?> blocked = exec.submit(new Runnable() {
            public void run() {
                try {
                    // Il semaforo impedisce un'ulteriore estrazione,
                    // quindi la chiamata e' bloccata:
                    pool.checkOut();
                } catch(InterruptedException e) {
                    print("checkOut() Interrupted");
                }
            }
        });
    }
}
```



```
TimeUnit.SECONDS.sleep(2);
blocked.cancel(true); // Uscita dalla chiamata bloccata
print("Checking in objects in " + list);
for(Fat f : list)
    pool.checkIn(f);
for(Fat f : list)
    pool.checkIn(f); // Secondo checkIn ignorato
exec.shutdown();
}
} /* (Da eseguire per visualizzare l'output) */:-
```

In **main()** viene creato un **Pool** per mantenere gli oggetti **Fat** e un insieme di **CheckoutTask** inizia a utilizzare il **Pool**. Poi il thread **main()** comincia a estrarre gli oggetti **Fat** e non li restituisce. Quando sono stati estratti tutti gli oggetti nel pool, **Semaphore** non ammette più estrazioni. Il metodo **run()** di **blocked** rimane quindi bloccato e dopo due secondi interviene il metodo **cancel()** per uscire da **Future**. Notate che il **Pool** ignora gli inserimenti ridondanti.

Questo esempio fa affidamento sulla scrupolosità del client del **Pool** e sul fatto che reinserisce volontariamente gli elementi. Nel caso non possiate sempre contare su queste certezze, in *Thinking in Patterns (with Java)* troverete ulteriori suggerimenti sui modi per gestire gli oggetti estratti dai pool: il volume è scaricabile dal sito [www.mindview.net](http://www.mindview.net).

### **Exchanger**

Un **Exchanger** rappresenta una barriera che scambia oggetti tra due task: quando i task entrano nella barriera possiedono un oggetto; quando escono dalla barriera, possiedono l'oggetto che in precedenza era detenuto dall'altro task. Di norma gli **Exchanger** vengono utilizzati quando un task crea oggetti la cui generazione è onerosa, e un altro task li “consuma”; in questo modo, più oggetti possono essere creati nello stesso momento in cui vengono utilizzati.

Per utilizzare la classe **Exchanger** creerete i task produttore e consumatore che, tramite i generici e **Generator**, lavoreranno con qualunque tipo di oggetto; poi applicherete tali task alla classe **Fat**.

I task **ExchangerProducer** ed **ExchangerConsumer** utilizzano una **List<T>** come oggetto da scambiare; ognuno di essi contiene un **Exchanger** per questa **List<T>**. Quando chiamate il metodo **Exchanger.exchange()**, esso si blocca



finché il task partner non chiama il suo metodo `exchange()`. Non appena entrambi i metodi `exchange()` sono stati completati, le due `List<T>` vengono permutate.

```
//: concurrency/ExchangerDemo.java
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class ExchangerProducer<T> implements Runnable {
    private Generator<T> generator;
    private Exchanger<List<T>> exchanger;
    private List<T> holder;
    ExchangerProducer(Exchanger<List<T>> exchg,
    Generator<T> gen, List<T> holder) {
        exchanger = exchg;
        generator = gen;
        this.holder = holder;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                for(int i = 0; i < ExchangerDemo.size; i++)
                    holder.add(generator.next());
                // Scambia pieno con vuoto:
                holder = exchanger.exchange(holder);
            }
        } catch(InterruptedException e) {
            // OK per uscire in questo modo
        }
    }
}

class ExchangerConsumer<T> implements Runnable {
    private Exchanger<List<T>> exchanger;
    private List<T> holder;
    private volatile T value;
```



```
ExchangerConsumer(Exchanger<List<T>> ex, List<T> holder){  
    exchanger = ex;  
    this.holder = holder;  
}  
public void run() {  
    try {  
        while(!Thread.interrupted()) {  
            holder = exchanger.exchange(holder);  
            for(T x : holder) {  
                value = x; // Estrae value  
                holder.remove(x); // OK per CopyOnWriteArrayList  
            }  
        }  
    } catch(InterruptedException e) {  
        // OK per uscire in questo modo  
    }  
    System.out.println("Final value: " + value);  
}  
}  
  
public class ExchangerDemo {  
    static int size = 10;  
    static int delay = 5; // Secondi  
    public static void main(String[] args) throws Exception {  
        if(args.length > 0)  
            size = new Integer(args[0]);  
        if(args.length > 1)  
            delay = new Integer(args[1]);  
        ExecutorService exec = Executors.newCachedThreadPool();  
        Exchanger<List<Fat>> xc = new Exchanger<List<Fat>>();  
        List<Fat>  
        producerList = new CopyOnWriteArrayList<Fat>(),  
        consumerList = new CopyOnWriteArrayList<Fat>();  
        exec.execute(new ExchangerProducer<Fat>(xc,  
            BasicGenerator.create(Fat.class), producerList));  
        exec.execute(  
            new ExchangerConsumer<Fat>(xc, consumerList));  
    }  
}
```



```
    TimeUnit.SECONDS.sleep(delay);
    exec.shutdownNow();
}
} /* Output: (Esempio)
Final value: Fat id: 29999
*///:~
```

In **main()** viene dapprima creato un solo oggetto **Exchanger** che sia utilizzabile da entrambi i task, poi i due oggetti **CopyOnWriteArrayList** per la permutazione.

Questa particolare variante di **List** ammette che sia chiamato il metodo **remove()** mentre la lista viene percorsa, senza sollevare un'eccezione **ConcurrentModificationException**. **ExchangerProducer** popola una **List**, poi la permuta con quella vuota che gli viene passata da **ExchangerConsumer**. Grazie a **Exchanger**, il popolamento di una lista e l'utilizzo dell'altra avvengono simultaneamente.

**Esercizio 34 (1)** Modificate **ExchangerDemo.java** per utilizzare una vostra classe anziché la classe **Fat**.

## Simulazione

Uno degli impieghi più interessanti della concorrenza è la creazione di simulazioni. Mediante la concorrenza ogni componente di una simulazione può essere rappresentato da un task, e questo semplifica notevolmente la realizzazione di un programma di simulazione. Molti videogiochi e animazioni CGI utilizzate nei film sono simulazioni; **HorseRace.java** e **GreenhouseScheduler.java**, mostrati in precedenza, sono anch'essi (lo ha detto esplicitamente) simulazioni.

### *Simulatore di uno sportello bancario*

Questa classica simulazione può rappresentare qualsiasi situazione in cui gli oggetti compaiono casualmente, e richiedono un periodo di tempo casuale per essere serviti da un numero limitato di server. È quindi possibile sviluppare la simulazione per determinare il numero ideale di server.

In questo esempio, per ogni cliente della banca è prevista una certa quantità di tempo di servizio. La quantità di tempo di servizio sarà diversa per ogni cliente e determinata a caso; inoltre non è noto a priori quanti clienti giun-



geranno in un certo intervallo di tempo, poiché anche questo valore sarà determinato in modo casuale.

```
//: concurrency/BankTellerSimulation.java
// Utilizzo di code e multithreading.
// {Args: 5}
import java.util.concurrent.*;
import java.util.*;

// Gli oggetti di sola lettura non richiedono sincronizzazione:
class Customer {
    private final int serviceTime;
    public Customer(int tm) { serviceTime = tm; }
    public int getServiceTime() { return serviceTime; }
    public String toString() {
        return "[" + serviceTime + "]";
    }
}

// Indica alla coda di clienti di visualizzarsi:
class CustomerLine extends ArrayBlockingQueue<Customer> {
    public CustomerLine(int maxLineSize) {
        super(maxLineSize);
    }
    public String toString() {
        if(this.size() == 0)
            return "[Empty]";
        StringBuilder result = new StringBuilder();
        for(Customer customer : this)
            result.append(customer);
        return result.toString();
    }
}

// Aggiunge clienti casuali a una coda:
class CustomerGenerator implements Runnable {
    private CustomerLine customers;
```



```
private static Random rand = new Random(47);
public CustomerGenerator(CustomerLine cq) {
    customers = cq;
}
public void run() {
    try {
        while(!Thread.interrupted()) {
            TimeUnit.MILLISECONDS.sleep(rand.nextInt(300));
            customers.put(new Customer(rand.nextInt(1000)));
        }
    } catch(InterruptedException e) {
        System.out.println("CustomerGenerator interrupted");
    }
    System.out.println("CustomerGenerator terminating");
}
}

class Teller implements Runnable, Comparable<Teller> {
    private static int counter = 0;
    private final int id = counter++;
    // Clienti serviti nel corso del turno:
    private int customersServed = 0;
    private CustomerLine customers;
    private boolean servingCustomerLine = true;
    public Teller(CustomerLine cq) { customers = cq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                Customer customer = customers.take();
                TimeUnit.MILLISECONDS.sleep(
                    customer.getServiceTime());
                synchronized(this) {
                    customersServed++;
                    while(!servingCustomerLine)
                        wait();
                }
            }
        }
    }
}
```



```
        } catch(InterruptedException e) {
            System.out.println(this + "interrupted");
        }
        System.out.println(this + "terminating");
    }

    public synchronized void doSomethingElse() {
        customersServed = 0;
        servingCustomerLine = false;
    }

    public synchronized void serveCustomerLine() {
        assert !servingCustomerLine:"already serving: " + this;
        servingCustomerLine = true;
        notifyAll();
    }

    public String toString() { return "Teller " + id + " "; }
    public String shortString() { return "T" + id; }
    // Utilizzato dalla coda prioritaria:
    public synchronized int compareTo(Teller other) {
        return customersServed < other.customersServed ? -1 :
               (customersServed == other.customersServed ? 0 : 1);
    }
}

class TellerManager implements Runnable {
    private ExecutorService exec;
    private CustomerLine customers;
    private PriorityQueue<Teller> workingTellers =
        new PriorityQueue<Teller>();
    private Queue<Teller> tellersDoingOtherThings =
        new LinkedList<Teller>();
    private int adjustmentPeriod;
    private static Random rand = new Random(47);
    public TellerManager(ExecutorService e,
                        CustomerLine customers, int adjustmentPeriod) {
        exec = e;
        this.customers = customers;
        this.adjustmentPeriod = adjustmentPeriod;
    }
}
```



```
// Inizia con un solo sportello:  
Teller teller = new Teller(customers);  
exec.execute(teller);  
workingTellers.add(teller);  
}  
public void adjustTellerNumber() {  
    // Questo e' l'effettivo sistema di controllo. Adattando  
    // questi valori potete evidenziare problemi di stabilita'  
    // nel meccanismo di controllo.  
    // Se la fila e' troppo lunga, attiva un altro sportello:  
    if(customers.size() / workingTellers.size() > 2) {  
        // Se i cassieri sono in pausa o impegnati in altre  
        // attivita', ne viene richiamato uno:  
        if(tellersDoingOtherThings.size() > 0) {  
            Teller teller = tellersDoingOtherThings.remove();  
            teller.serveCustomerLine();  
            workingTellers.offer(teller);  
            return;  
        }  
        // In caso contrario crea (mette in servizio) un nuovo  
        // sportello  
        Teller teller = new Teller(customers);  
        exec.execute(teller);  
        workingTellers.add(teller);  
        return;  
    }  
    // Se la fila e' abbastanza corta, elimina uno sportello:  
    if(workingTellers.size() > 1 &&  
        customers.size() / workingTellers.size() < 2)  
        reassignOneTeller();  
    // Se non c'e' nessuna coda, basta un solo sportello:  
    if(customers.size() == 0)  
        while(workingTellers.size() > 1)  
            reassignOneTeller();  
    }  
    // Assegna a un cassiere un'altra attivita' o lo mette in  
    // pausa:
```



```
private void reassignOneTeller() {
    Teller teller = workingTellers.poll();
    teller.doSomethingElse();
    tellersDoingOtherThings.offer(teller);
}
public void run() {
    try {
        while(!Thread.interrupted()) {
            TimeUnit.MILLISECONDS.sleep(adjustmentPeriod);
            adjustTellerNumber();
            System.out.print(customers + " { ");
            for(Teller teller : workingTellers)
                System.out.print(teller.shortString() + " ");
            System.out.println("}");
        }
    } catch(InterruptedException e) {
        System.out.println(this + " interrupted");
    }
    System.out.println(this + " terminating");
}
public String toString() { return "TellerManager ";}
}

public class BankTellerSimulation {
    static final int MAX_LINE_SIZE = 50;
    static final int ADJUSTMENT_PERIOD = 1000;
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        // Se la coda e' troppo lunga i clienti se ne andranno:
        CustomerLine customers =
            new CustomerLine(MAX_LINE_SIZE);
        exec.execute(new CustomerGenerator(customers));
        // Il gestore aggiunge o elimina gli sportelli quando
        // necessario:
        exec.execute(new TellerManager(
            exec, customers, ADJUSTMENT_PERIOD));
    }
}
```



```
if(args.length > 0) // Argomento opzionale
    TimeUnit.SECONDS.sleep(new Integer(args[0]));
else {
    System.out.println("Press 'Enter' to quit");
    System.in.read();
}
exec.shutdownNow();
}

} /* Output: (Esempio)
[200][207] { T1 T0 }
[861][258][140][322] { T1 T0 }
[575][342][804][826][896][984] { T2 T0 T1 }
[984][810][141][12][689][992][976][368][395][354] { T3 T2 T1
T0 }

Teller 2 interrupted
Teller 2 terminating
Teller 1 interrupted
Teller 1 terminating
TellerManager interrupted
TellerManager terminating
Teller 3 interrupted
Teller 3 terminating
Teller 0 interrupted
Teller 0 terminating
CustomerGenerator interrupted
CustomerGenerator terminating
*///:~
```

Gli oggetti **Customer** sono molto semplici, dal momento che contengono soltanto un campo **final int**. Poiché sono immutabili, questi oggetti sono di sola lettura e non necessitano né della sincronizzazione né dell'utilizzo di **volatile**. Oltre a questo, ogni task **Teller** rimuove dalla coda di input un solo **Customer** per volta e lavora con quel **Customer** fino al completamento del servizio, in modo che a un **Customer** acceda comunque un solo task per volta.

**CustomerLine** rappresenta una fila in cui i clienti attendono prima di essere serviti da uno sportello **Teller**: si tratta semplicemente di un'**ArrayBlockingQueue** che possiede un metodo **toString()** per visualizzare i risultati nel modo voluto.



Un **CustomerGenerator** è connesso a una **CustomerLine** e mette i **Customer** in coda, secondo intervalli casuali.

Un **Teller** “preleva” i **Customer** dalla **CustomerLine** e li elabora uno per volta, tenendo traccia del numero di **Customer** serviti in un determinato turno. Potete ordinare al cassiere dello sportello di eseguire “qualcos’altro” (**doSomethingElse()**) quando non vi sono clienti in fila e di servirli con **serveCustomerLine()** qualora ve ne siano. Per scegliere il successivo sportello da rimettere in servizio il metodo **compareTo()** esamina il numero di clienti serviti, in modo che una **PriorityQueue** possa mettere automaticamente in prima posizione lo sportello meno operativo.

**TellerManager** è il fulcro dell’attività, poiché tiene traccia di tutti gli sportelli e della situazione dei clienti. Uno degli aspetti interessanti di questa simulazione è che cerca di determinare il numero ottimale di cassieri a fronte del flusso di clienti. Questo è evidente nel metodo **adjustTellerNumber()**, che è un sistema di controllo per abilitare e rendere non operativi gli sportelli, caratterizzato da stabilità. Tutti i sistemi di controllo, infatti, presentano problemi in questo senso: se reagiscono troppo rapidamente a un cambiamento diventano instabili, e se reagiscono troppo lentamente il sistema si sposta verso uno dei suoi estremi.

**Esercizio 35 (8)** Modificate **BankTellerSimulation.java** in modo che simuli gli utenti web che inoltrano le loro richieste a un numero fisso di server: l’obiettivo è determinare il carico di lavoro che il gruppo di server è in grado di gestire.

### ***Simulatore di un ristorante***

Questa simulazione ha origine dal semplice esempio **Restaurant.java** presentato in precedenza, cui sono stati aggiunti altri componenti di simulazione, come gli ordini (**Order**) e le portate (**Plates**); il test riutilizza le classi **menu** che sono state presentate nel Volume 2, Capitolo 7.

La simulazione introduce anche l’utilizzo del componente **SynchronousQueue** di Java SE5: si tratta di una coda bloccante priva di capienza interna, pertanto ogni chiamata a **put()** deve corrispondere a una chiamata a **take()** e viceversa. È come se passaste un oggetto a qualcuno: non esiste un tavolo su cui appoggiare l’oggetto, quindi il passaggio funziona unicamente se l’altra persona porge la mano, pronta a ricevere l’oggetto. In questo esempio la **SynchronousQueue** rappresenta un punto qualsiasi di fronte al commensale, per rafforzare il concetto che può essere servita soltanto una portata per volta.



Le altre classi e funzionalità di questo esempio derivano dalla struttura di **Restaurant.java** o corrispondono in modo ragionevolmente credibile al funzionamento di un vero ristorante.

```
//: concurrency/restaurant2/RestaurantWithQueues.java
// {Args: 5}
package concurrency.restaurant2;
import enumerated.menu.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Questo e' l'ordine passato al cameriere, che lo trasmette
// allo chef:
class Order { // (Un oggetto di trasferimento dati)
    private static int counter = 0;
    private final int id = counter++;
    private final Customer customer;
    private final WaitPerson waitPerson;
    private final Food food;
    public Order(Customer cust, WaitPerson wp, Food f) {
        customer = cust;
        waitPerson = wp;
        food = f;
    }
    public Food item() { return food; }
    public Customer getCustomer() { return customer; }
    public WaitPerson getWaitPerson() { return waitPerson; }
    public String toString() {
        return "Order: " + id + " item: " + food +
            " for: " + customer +
            " served by: " + waitPerson;
    }
}

// Questa e' la portata che viene restituita dallo chef:
class Plate {
    private final Order order;
```



```
private final Food food;
public Plate(Order ord, Food f) {
    order = ord;
    food = f;
}
public Order getOrder() { return order; }
public Food getFood() { return food; }
public String toString() { return food.toString(); }
}

class Customer implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final WaitPerson waitPerson;
    // Puo' essere ricevuta solo una portata per volta:
    private SynchronousQueue<Plate> placeSetting =
        new SynchronousQueue<Plate>();
    public Customer(WaitPerson w) { waitPerson = w; }
    public void
    deliver(Plate p) throws InterruptedException {
        // Si blocca soltanto se il cliente sta ancora mangiando
        // la portata precedente:
        placeSetting.put(p);
    }
    public void run() {
        for(Course course : Course.values()) {
            Food food = course.randomSelection();
            try {
                waitPerson.placeOrder(this, food);
                // Si blocca finche' la portata non e' stata consegnata:
                print(this + "eating " + placeSetting.take());
            } catch(InterruptedException e) {
                print(this + "waiting for " +
                    course + " interrupted");
                break;
            }
        }
    }
}
```



```
        print(this + "finished meal, leaving");
    }
    public String toString() {
        return "Customer " + id + " ";
    }
}

class WaitPerson implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final Restaurant restaurant;
    BlockingQueue<Plate> filledOrders =
        new LinkedBlockingQueue<Plate>();
    public WaitPerson(Restaurant rest) { restaurant = rest; }
    public void placeOrder(Customer cust, Food food) {
        try {
            // In realta' non dovrebbe bloccarsi perche' questa e'
            // una LinkedBlockingQueue senza limiti di dimensioni:
            restaurant.orders.put(new Order(cust, this, food));
        } catch(InterruptedException e) {
            print(this + " placeOrder interrupted");
        }
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Si blocca finche' la portata e' pronta
                Plate plate = filledOrders.take();
                print(this + "received " + plate +
                      " delivering to " +
                      plate.getOrder().getCustomer());
                plate.getOrder().getCustomer().deliver(plate);
            }
        } catch(InterruptedException e) {
            print(this + " interrupted");
        }
    }
}
```



```
    print(this + " off duty");
}
public String toString() {
    return "WaitPerson " + id + " ";
}
}

class Chef implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final Restaurant restaurant;
    private static Random rand = new Random(47);
    public Chef(Restaurant rest) { restaurant = rest; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Si blocca finche' non arriva un ordine:
                Order order = restaurant.orders.take();
                Food requestedItem = order.item();
                // Tempo di preparazione dell'ordine:
                TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
                Plate plate = new Plate(order, requestedItem);
                order.getWaitPerson().filledOrders.put(plate);
            }
        } catch(InterruptedException e) {
            print(this + " interrupted");
        }
        print(this + " off duty");
    }
    public String toString() { return "Chef " + id + " "; }
}

class Restaurant implements Runnable {
    private List<WaitPerson> waitPersons =
        new ArrayList<WaitPerson>();
    private List<Chef> chefs = new ArrayList<Chef>();
    private ExecutorService exec;
```



```
private static Random rand = new Random(47);
BlockingQueue<Order>
    orders = new LinkedBlockingQueue<Order>();
public Restaurant(ExecutorService e, int nWaitPersons,
    int nChefs) {
    exec = e;
    for(int i = 0; i < nWaitPersons; i++) {
        WaitPerson waitPerson = new WaitPerson(this);
        waitPersons.add(waitPerson);
        exec.execute(waitPerson);
    }
    for(int i = 0; i < nChefs; i++) {
        Chef chef = new Chef(this);
        chefs.add(chef);
        exec.execute(chef);
    }
}
public void run() {
    try {
        while(!Thread.interrupted()) {
            // Arriva un nuovo cliente e gli viene assegnato
            // un cameriere WaitPerson:
            WaitPerson wp = waitPersons.get(
                rand.nextInt(waitPersons.size()));
            Customer c = new Customer(wp);
            exec.execute(c);
            TimeUnit.MILLISECONDS.sleep(100);
        }
    } catch(InterruptedException e) {
        print("Restaurant interrupted");
    }
    print("Restaurant closing");
}
}

public class RestaurantWithQueues {
    public static void main(String[] args) throws Exception {
```



```
ExecutorService exec = Executors.newCachedThreadPool();
Restaurant restaurant = new Restaurant(exec, 5, 2);
exec.execute(restaurant);
if(args.length > 0) // Argomento opzionale
    TimeUnit.SECONDS.sleep(new Integer(args[0]));
else {
    print("Press 'Enter' to quit");
    System.in.read();
}
exec.shutdownNow();
}

} /* Output: (Esempio)
WaitPerson 0 received SPRING_ROLLS delivering to Customer 1
Customer 1 eating SPRING_ROLLS
WaitPerson 3 received SPRING_ROLLS delivering to Customer 0
Customer 0 eating SPRING_ROLLS
WaitPerson 0 received BURRITO delivering to Customer 1
Customer 1 eating BURRITO
WaitPerson 3 received SPRING_ROLLS delivering to Customer 2
Customer 2 eating SPRING_ROLLS
WaitPerson 1 received SOUP delivering to Customer 3
Customer 3 eating SOUP
WaitPerson 3 received VINDALOO delivering to Customer 0
Customer 0 eating VINDALOO
WaitPerson 0 received FRUIT delivering to Customer 1
...
*///:-
```

Un particolare molto importante da notare in questo esempio è che la gestione della complessità avviene mediante code che comunicano tra i task. Questa tecnica, da sola, facilita notevolmente il processo di programmazione concorrente invertendo il controllo: i task non interferiscono direttamente l'uno con l'altro, ma si trasmettono l'un l'altro gli oggetti per mezzo delle code. Il task ricevente gestisce l'oggetto considerandolo come un messaggio vero e proprio, senza assegnargli un significato implicito. Se adottate questa tecnica ogni volta che vi è possibile avrete maggiori probabilità di realizzare applicazioni concorrenti robuste e affidabili.



**Esercizio 36 (10)** Modificate **RestaurantWithQueues.java** in modo che esista un oggetto **OrderTicket** per emulare le ordinazioni e le precedenze di ogni tavolo. Modificate **order** in **orderTicket** e aggiungete una classe **Table** per gestire i tavoli, con più **Customer** per ogni tavolo.

### Attività distribuite

Quello che segue è un esempio di simulazione che riepiloga molti dei concetti presentati in questo capitolo, considerando un'ipotetica catena di montaggio robotizzata per la produzione di automobili. Ogni automobile (**Car**) viene costruita in più fasi, iniziando dalla creazione del telaio, seguita dal montaggio del motore, della trasmissione e delle ruote.

```
//: concurrency/CarBuilder.java
// Un esempio complesso di task cooperativi.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Car {
    private final int id;
    private boolean
        engine = false, driveTrain = false, wheels = false;
    public Car(int idn) { id = idn; }
    // Oggetto Car vuoto:
    public Car() { id = -1; }
    public synchronized int getId() { return id; }
    public synchronized void addEngine() { engine = true; }
    public synchronized void addDriveTrain() {
        driveTrain = true;
    }
    public synchronized void addWheels() { wheels = true; }
    public synchronized String toString() {
        return "Car " + id + " [" + " engine: " + engine
            + " driveTrain: " + driveTrain
            + " wheels: " + wheels + " ]";
    }
}
```



```
class CarQueue extends LinkedBlockingQueue<Car> {}

class ChassisBuilder implements Runnable {
    private CarQueue carQueue;
    private int counter = 0;
    public ChassisBuilder(CarQueue cq) { carQueue = cq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(500);
                // Costruisce il telaio:
                Car c = new Car(counter++);
                print("ChassisBuilder created " + c);
                // Inserisce nella coda
                carQueue.put(c);
            }
        } catch(InterruptedException e) {
            print("Interrupted: ChassisBuilder");
        }
        print("ChassisBuilder off");
    }
}

class Assembler implements Runnable {
    private CarQueue chassisQueue, finishingQueue;
    private Car car;
    private CyclicBarrier barrier = new CyclicBarrier(4);
    private RobotPool robotPool;
    public Assembler(CarQueue cq, CarQueue fq, RobotPool rp){
        chassisQueue = cq;
        finishingQueue = fq;
        robotPool = rp;
    }
    public Car car() { return car; }
    public CyclicBarrier barrier() { return barrier; }
    public void run() {
        try {
```



```
while(!Thread.interrupted()) {  
    // Si blocca finche' il telaio non e' disponibile:  
    car = chassisQueue.take();  
    // Mette in servizio alcuni robot per eseguire il  
    // lavoro:  
    robotPool.hire(EngineRobot.class, this);  
    robotPool.hire(DriveTrainRobot.class, this);  
    robotPool.hire(WheelRobot.class, this);  
    barrier.await(); // Fino a quando i robot terminano  
    // Inserisce l'automobile in finishingQueue per  
    // ulteriori lavorazioni  
    finishingQueue.put(car);  
}  
}  
} catch(InterruptedException e) {  
    print("Exiting Assembler via interrupt");  
} catch(BrokenBarrierException e) {  
    // Di questo sarebbe interessante sapere qualcosa di piu'  
    throw new RuntimeException(e);  
}  
print("Assembler off");  
}  
}  
  
class Reporter implements Runnable {  
    private CarQueue carQueue;  
    public Reporter(CarQueue cq) { carQueue = cq; }  
    public void run() {  
        try {  
            while(!Thread.interrupted()) {  
                print(carQueue.take());  
            }  
        } catch(InterruptedException e) {  
            print("Exiting Reporter via interrupt");  
        }  
        print("Reporter off");  
    }  
}
```



```
abstract class Robot implements Runnable {
    private RobotPool pool;
    public Robot(RobotPool p) { pool = p; }
    protected Assembler assembler;
    public Robot assignAssembler(Assembler assembler) {
        this.assembler = assembler;
        return this;
    }
    private boolean engage = false;
    public synchronized void engage() {
        engage = true;
        notifyAll();
    }
    // La parte di run() che e' diversa per ogni robot:
    abstract protected void performService();
    public void run() {
        try {
            powerDown(); // Attende per il tempo necessario
            while(!Thread.interrupted()) {
                performService();
                assembler.barrier().await(); // Sincronizza
                // Questo lavoro e' terminato...
                powerDown();
            }
        } catch(InterruptedException e) {
            print("Exiting " + this + " via interrupt");
        } catch(BrokenBarrierException e) {
            // Di questo sarebbe interessante sapere qualcosa di piu'
            throw new RuntimeException(e);
        }
        print(this + " off");
    }
    private synchronized void
powerDown() throws InterruptedException {
    engage = false;
    assembler = null; // Scollega dall'Assembler
    // Reinseririsce nel pool disponibile:
```



```
pool.release(this);
while(engage == false) // Spegnimento
    wait();
}
public String toString() { return getClass().getName(); }
}

class EngineRobot extends Robot {
    public EngineRobot(RobotPool pool) { super(pool); }
    protected void performService() {
        print(this + " installing engine");
        assembler.car().addEngine();
    }
}

class DriveTrainRobot extends Robot {
    public DriveTrainRobot(RobotPool pool) { super(pool); }
    protected void performService() {
        print(this + "installing DriveTrain");
        assembler.car().addDriveTrain();
    }
}

class WheelRobot extends Robot {
    public WheelRobot(RobotPool pool) { super(pool); }
    protected void performService() {
        print(this + "installing Wheels");
        assembler.car().addWheels();
    }
}

class RobotPool {
    // Evita tacitamente voci identiche:
    private Set<Robot> pool = new HashSet<Robot>();
    public synchronized void add(Robot r) {
        pool.add(r);
        notifyAll();
    }
}
```



```
public synchronized void
hire(Class<? extends Robot> robotType, Assembler d)
throws InterruptedException {
    for(Robot r : pool)
        if(r.getClass().equals(robotType)) {
            pool.remove(r);
            r.assignAssembler(d);
            r.engage(); // Attiva per eseguire il lavoro
            return;
        }
    wait(); // Nessuno disponibile
    hire(robotType, d); // Riprova piu' volte
}
public synchronized void release(Robot r) { add(r); }

public class CarBuilder {
    public static void main(String[] args) throws Exception {
        CarQueue chassisQueue = new CarQueue(),
                    finishingQueue = new CarQueue();
        ExecutorService exec = Executors.newCachedThreadPool();
        RobotPool robotPool = new RobotPool();
        exec.execute(new EngineRobot(robotPool));
        exec.execute(new DriveTrainRobot(robotPool));
        exec.execute(new WheelRobot(robotPool));
        exec.execute(new Assembler(
            chassisQueue, finishingQueue, robotPool));
        exec.execute(new Reporter(finishingQueue));
        // Inizia l'esecuzione producendo il telaio:
        exec.execute(new ChassisBuilder(chassisQueue));
        TimeUnit.SECONDS.sleep(7);
        exec.shutdownNow();
    }
} /* (Da eseguire per visualizzare l'output) *///:~
```

Gli oggetti **Car** sono trasportati da un punto a un altro della linea di produzione tramite un oggetto **CarQueue**, che è un tipo di **LinkedBlockingQueue**.



Un **ChassisBuilder** genera il telaio di un'automobile (**Car**) e lo dispone sulla **CarQueue**. L'**Assembler** preleva **Car** da **CarQueue** e mette in servizio alcuni **Robot** per lavorare all'automobile.

Un oggetto **CyclicBarrier** consente ad **Assembler** di attendere fino a quando tutti i **Robot** abbiano terminato il loro lavoro, poi inserisce nuovamente la **Car** sulla **CarQueue** uscente, per trasportare l'automobile alla stazione di lavoro successiva. Il consumatore della **CarQueue** finale è un oggetto **Reporter**, che si limita a visualizzare **Car** per dimostrare che i task sono stati eseguiti correttamente.

I **Robot** vengono gestiti in un pool: quando occorre eseguire un'operazione il **Robot** viene messo in servizio prelevandolo dal pool, in cui ritorna dopo avere terminato il proprio lavoro.

In **main()** vengono creati tutti gli oggetti necessari e i task inizializzati; **ChassisBuilder** viene avviato per ultimo e inizia il processo di produzione. Tenete presente che, a causa del comportamento della classe **LinkedBlockingQueue**, **ChassisBuilder** potrebbe anche essere avviato per primo. Notate che questo programma segue le linee guida relative al ciclo di vita dei task e degli oggetti presentate in questo capitolo, quindi il processo di arresto dei vari componenti avviene in modo sicuro.

Notate che i metodi di **Car** sono tutti **synchronized**. In questo esempio specifico ciò è ridondante, in quanto nella fabbrica simulata le **Car** si muovono per mezzo di code e solo un task per volta può lavorare su un oggetto **Car**; in pratica, le code forzano l'accesso serializzato agli oggetti **Car**. Tuttavia questo è esattamente il tipo di trappola in cui è facile cadere: dal momento che in questo caso **synchronized** sembra inutile, potreste ritenere possibile ottimizzare l'applicazione evitando di sincronizzare la classe **Car**. In seguito, però, qualora il sistema fosse collegato a un altro che richiede un oggetto **Car synchronized** si verificherebbero malfunzionamenti.

In proposito, Brian Goetz commenta:

*"è molto più semplice affermare che, tenuto conto che l'oggetto **Car** potrebbe essere utilizzato da vari thread, sarebbe opportuno renderlo thread-safe nel modo consueto. Ritengo che questo approccio sia ben rappresentato dalla seguente analogia. Nei parchi naturali, dove il terreno è molto scosceso è normale trovare steccati, recinzioni o protezioni segnalati con cartelli del tipo 'Vietato sporgersi'. Naturalmente il vero obiettivo di questo divieto non è impedirvi di appoggiarvi alla recinzione, bensì di cadere dalla scarpata o burrone che sia. Tuttavia 'Vietato sporgersi' è una regola molto più semplice da seguire rispetto a 'Vietato cadere nel burrone'."*



**Esercizio 37** (2) Modificate `CarBuilder.java` per aggiungere un'altra fase al processo di produzione dell'automobile, in cui saranno montati il sistema di scarico, la carrozzeria e gli ammortizzatori. Poi fate in modo che questi processi possano essere simultaneamente eseguiti da robot.

**Esercizio 38** (3) Adottando un approccio analogo a `CarBuilder.java` realizzate una simulazione della costruzione di una casa, di cui si è accennato nella prima parte di questo capitolo.

## Ottimizzazione delle prestazioni

La libreria `java.util.concurrent` di Java SE5 contiene numerose classi specificamente utili per migliorare le prestazioni. Analizzando la libreria `concurrent` non è sempre facile individuare quali classi siano destinate a un utilizzo normale (`BlockingQueue`, per esempio) e quali siano specifiche per l'ottimizzazione delle prestazioni. Nei prossimi paragrafi potrete esaminare alcuni dei problemi e delle classi che hanno attinenza con questo argomento.

### Confronto tra le tecnologie mutex

Ora che Java include sia la vecchia parola chiave `synchronized` sia le nuove classi `Lock` e `Atomic` di Java SE5, è interessante confrontare i diversi criteri in modo da comprenderne meglio il valore e le modalità di utilizzo.

Il metodo più banale da seguire per raggiungere questo scopo consiste semplicemente nel testare ogni tecnica, come mostrato di seguito.

```
//: concurrency/SimpleMicroBenchmark.java
// I pericoli del microbenchmarking.
import java.util.concurrent.locks.*;

abstract class Incrementable {
    protected long counter = 0;
    public abstract void increment();
}

class SynchronizingTest extends Incrementable {
    public synchronized void increment() { ++counter; }
}
```



```
class LockingTest extends Incrementable {
    private Lock lock = new ReentrantLock();
    public void increment() {
        lock.lock();
        try {
            ++counter;
        } finally {
            lock.unlock();
        }
    }
}

public class SimpleMicroBenchmark {
    static long test(Incrementable incr) {
        long start = System.nanoTime();
        for(long i = 0; i < 10000000L; i++)
            incr.increment();
        return System.nanoTime() - start;
    }
    public static void main(String[] args) {
        long synchTime = test(new SynchronizingTest());
        long lockTime = test(new LockingTest());
        System.out.printf("synchronized: %1$10d\n", synchTime);
        System.out.printf("Lock:           %1$10d\n", lockTime);
        System.out.printf("Lock/synchronized = %1$.3f",
                         (double)lockTime/(double)synchTime);
    }
} /* Output: (75% match)
synchronized: 244919117
Lock:           939098964
Lock/synchronized = 3.834
*///:~
```

Come potete vedere dall'output, le chiamate al metodo **synchronized** sembrano essere più veloci rispetto all'utilizzo di un **ReentrantLock**. Che cosa è accaduto?



Questo esempio evidenzia i pericoli del cosiddetto “microbenchmarking”.<sup>19</sup>

Di solito questo termine si riferisce ai test prestazionali di una funzionalità “in isolamento”, fuori del contesto. Naturalmente, per verificare asserzioni quali “**Lock** è molto più veloce di **synchronized**” dovrete sempre scrivere test, tuttavia nel momento in cui lo farete dovete essere consapevoli di quanto accade realmente durante la compilazione e l'esecuzione.

L'esempio **SimpleMicroBenchmark.java** rileva un certo numero di problemi. Innanzitutto, le effettive differenze di prestazioni si rendono evidenti soltanto quando i mutex sono in conflitto, pertanto devono esistere diversi task che cercano di accedere alle sezioni di codice oggetto di mutex. Nell'esempio precedente, invece, ogni mutex è analizzato dal singolo thread **main()**, in condizioni di isolamento.

In secondo luogo è possibile che il compilatore esegua ottimizzazioni speciali in presenza della parola chiave **synchronized**, rilevando in alcuni casi anche che il programma è a thread singolo. Il compilatore potrebbe persino identificare che il **counter** viene incrementato soltanto un numero fisso di volte, limitandosi a pre-calcolare il risultato. Diversi compilatori e sistemi di runtime hanno comportamenti variabili, ed è quindi difficile sapere esattamente che cosa accadrà: in ogni caso occorre poter escludere la possibilità che il compilatore predica il risultato.

Per creare un test valido dovete rendere più complesso il programma. Per prima cosa vi occorrono vari task, non soltanto quelli che modificano i valori interni, ma anche task che leggano tali valori: in caso contrario la funzionalità di ottimizzazione del compilatore potrebbe riconoscere che i valori non vengono mai utilizzati. Inoltre, il calcolo deve essere abbastanza articolato e imprevedibile da fare in modo che il compilatore non possa procedere a ottimizzazioni “aggressive”. Questa condizione verrà soddisfatta precaricando un grande array di **int** casuali (precaricati al fine di ridurre l'impatto delle chiamate a **Random.nextInt()** sui cicli principali), e utilizzando tali valori in una sommatoria.

```
//: concurrency/SynchronizationComparisons.java
// Confronto tra le prestazioni di Lock e Atomic espliciti e
// quelle della parola chiave synchronized.
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
```

19. Brian Goetz si è rivelato prezioso nell'illustrare questi problemi all'autore; per maggiori dettagli sulla misurazione delle prestazioni, potete consultare l'articolo di Goetz, all'indirizzo [www-128.ibm.com/developerworks/library/j-jtp12214](http://www-128.ibm.com/developerworks/library/j-jtp12214).



```
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

abstract class Accumulator {
    public static long cycles = 50000L;
    // Numero di Modifier e Reader durante ogni test:
    private static final int N = 4;
    public static ExecutorService exec =
        Executors.newFixedThreadPool(N*2);
    private static CyclicBarrier barrier =
        new CyclicBarrier(N*2 + 1);
    protected volatile int index = 0;
    protected volatile long value = 0;
    protected long duration = 0;
    protected String id = "error";
    protected final static int SIZE = 100000;
    protected static int[] preLoaded = new int[SIZE];
    static {
        // Carica l'array di numeri casuali:
        Random rand = new Random(47);
        for(int i = 0; i < SIZE; i++)
            preLoaded[i] = rand.nextInt();
    }
    public abstract void accumulate();
    public abstract long read();
    private class Modifier implements Runnable {
        public void run() {
            for(long i = 0; i < cycles; i++)
                accumulate();
            try {
                barrier.await();
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```



```
private class Reader implements Runnable {
    private volatile long value;
    public void run() {
        for(long i = 0; i < cycles; i++)
            value = read();
        try {
            barrier.await();
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}
public void timedTest() {
    long start = System.nanoTime();
    for(int i = 0; i < N; i++) {
        exec.execute(new Modifier());
        exec.execute(new Reader());
    }
    try {
        barrier.await();
    } catch(Exception e) {
        throw new RuntimeException(e);
    }
    duration = System.nanoTime() - start;
    printf("%-13s: %13d\n", id, duration);
}
public static void
report(Accumulator acc1, Accumulator acc2) {
    printf("%-22s: %.2f\n", acc1.id + "/" + acc2.id,
           (double)acc1.duration/(double)acc2.duration);
}
}

class BaseLine extends Accumulator {
    { id = "BaseLine"; }
    public void accumulate() {
        value += preLoaded[index++];
    }
}
```



```
        if(index >= SIZE) index = 0;
    }
    public long read() { return value; }
}

class SynchronizedTest extends Accumulator {
    { id = "synchronized"; }
    public synchronized void accumulate() {
        value += preLoaded[index++];
        if(index >= SIZE) index = 0;
    }
    public synchronized long read() {
        return value;
    }
}

class LockTest extends Accumulator {
    { id = "Lock"; }
    private Lock lock = new ReentrantLock();
    public void accumulate() {
        lock.lock();
        try {
            value += preLoaded[index++];
            if(index >= SIZE) index = 0;
        } finally {
            lock.unlock();
        }
    }
    public long read() {
        lock.lock();
        try {
            return value;
        } finally {
            lock.unlock();
        }
    }
}
```



```
class AtomicTest extends Accumulator {
    { id = "Atomic"; }
    private AtomicInteger index = new AtomicInteger(0);
    private AtomicLong value = new AtomicLong(0);
    public void accumulate() {
        // Oops! Non si puo' utilizzare piu' di un Atomic per volta.
        // In ogni caso rimane pur sempre un indicatore di
        // prestazioni:
        int i = index.getAndIncrement();
        value.getAndAdd(preLoaded[i]);
        if(++i >= SIZE)
            index.set(0);
    }
    public long read() { return value.get(); }
}

public class SynchronizationComparisons {
    static BaseLine baseLine = new BaseLine();
    static SynchronizedTest synch = new SynchronizedTest();
    static LockTest lock = new LockTest();
    static AtomicTest atomic = new AtomicTest();
    static void test() {
        print("=====");
        printf("%-12s : %13d\n", "Cycles", Accumulator.cycles);
        baseLine.timedTest();
        synch.timedTest();
        lock.timedTest();
        atomic.timedTest();
        Accumulator.report(synch, baseLine);
        Accumulator.report(lock, baseLine);
        Accumulator.report(atomic, baseLine);
        Accumulator.report(synch, lock);
        Accumulator.report(synch, atomic);
        Accumulator.report(lock, atomic);
    }
    public static void main(String[] args) {
        int iterations = 5; // Predefinito
```



```
if(args.length > 0) // Cambia le iterazioni opzionalmente
    iterations = new Integer(args[0]);
// Popola il pool di thread la prima volta:
print("Warmup");
baseLine.timedTest();
// Ora il test iniziale non tiene conto dell'onere
// di avvio iniziale dei thread.
// Produce vari datapoint:
for(int i = 0; i < iterations; i++) {
    test();
    Accumulator.cycles *= 2;
}
Accumulator.exec.shutdown();
}
} /* Output: (Esempio)
Warmup
Baseline      :      34237033
=====
Cycles      :      50000
Baseline      :      20966632
synchronized :      24326555
Lock          :      53669950
Atomic        :      30552487
synchronized/BaseLine : 1.16
Lock/BaseLine      : 2.56
Atomic/BaseLine      : 1.46
synchronized/Lock      : 0.45
synchronized/Atomic      : 0.79
Lock/Atomic      : 1.76
=====
Cycles      :      100000
Baseline      :      41512818
synchronized :      43843003
Lock          :      87430386
Atomic        :      51892350
synchronized/BaseLine : 1.06
Lock/BaseLine      : 2.11
```



```
Atomic/BaseLine      : 1.25
synchronized/Lock    : 0.50
synchronized/Atomic   : 0.84
Lock/Atomic          : 1.68
=====
Cycles      : 200000
Baseline     : 80176670
synchronized : 5455046661
Lock         : 177686829
Atomic       : 101789194
synchronized/BaseLine : 68.04
Lock/BaseLine      : 2.22
Atomic/BaseLine     : 1.27
synchronized/Lock   : 30.70
synchronized/Atomic  : 53.59
Lock/Atomic         : 1.75
=====
Cycles      : 400000
Baseline     : 160383513
synchronized : 780052493
Lock         : 362187652
Atomic       : 202030984
synchronized/BaseLine : 4.86
Lock/BaseLine      : 2.26
Atomic/BaseLine     : 1.26
synchronized/Lock   : 2.15
synchronized/Atomic  : 3.86
Lock/Atomic         : 1.79
=====
Cycles      : 800000
Baseline     : 322064955
synchronized : 336155014
Lock         : 704615531
Atomic       : 393231542
synchronized/BaseLine : 1.04
Lock/BaseLine      : 2.19
Atomic/BaseLine     : 1.22
```



```
synchronized/Lock      : 0.47
synchronized/Atomic    : 0.85
Lock/Atomic           : 1.79
=====
Cycles      : 1600000
Baseline    : 650004120
synchronized : 52235762925
Lock        : 1419602771
Atomic      : 796950171
synchronized/BaseLine : 80.36
Lock/BaseLine       : 2.18
Atomic/BaseLine     : 1.23
synchronized/Lock   : 36.80
synchronized/Atomic  : 65.54
Lock/Atomic         : 1.78
=====
Cycles      : 3200000
Baseline    : 1285664519
synchronized : 96336767661
Lock        : 2846988654
Atomic      : 1590545726
synchronized/BaseLine : 74.93
Lock/BaseLine       : 2.21
Atomic/BaseLine     : 1.24
synchronized/Lock   : 33.84
synchronized/Atomic  : 60.57
Lock/Atomic         : 1.79
*///:~
```

Questo programma utilizza il design pattern *Template Method* per inserire tutto il codice comune nella classe di base, isolando il codice variabile nelle implementazioni di **accumulate()** e **read()** delle classi derivate.<sup>20</sup>

Nelle classi derivate **SynchronizedTest**, **LockTest** e **AtomicTest** potete vedere come **accumulate()** e **read()** esprimano i diversi modi per implementare l'esclusione reciproca.

20. Per maggiori informazioni su Template Method, consultate *Thinking in Patterns (with Java)* all'indirizzo [www.mindview.net](http://www.mindview.net).



In questo programma i task vengono eseguiti mediante un **FixedThreadPool** allo scopo di contenere la creazione dei thread all'inizio e impedire un carico di lavoro supplementare in fase di test. Per garantire che questo avvenga, il test iniziale viene duplicato e il primo risultato viene scartato perché include la creazione iniziale dei thread.

È necessario ricorrere a una **CyclicBarrier** per assicurarsi che tutti i task siano terminati prima di dichiarare completo ogni test.

Una clausola **static** precarica l'array di numeri casuali, prima che tutti i test comincino; in questo modo, se la generazione di questi numeri comportasse oneri aggiuntivi il test non ne terrebbe conto.

Ogni volta che viene chiamato **accumulate()**, esso si sposta nella posizione successiva dell'array **preLoaded** (a partire all'inizio dell'array) e aggiunge un numero casuale a **value**. I task multipli di **Modifier** e **Reader** forniscono le condizioni di conflitto all'oggetto **Accumulator**.

Notate che in **AtomicTest** la situazione è troppo complessa per provare a utilizzare gli oggetti di **Atomic**; in pratica, se sono coinvolti più oggetti **Atomic** probabilmente sarete costretti a rinunciare e tornare all'utilizzo dei mutex, più convenzionali: infatti, la documentazione JDK specifica che gli oggetti **Atomic** funzionano soltanto quando gli aggiornamenti critici di un oggetto sono limitati a una singola variabile. In ogni caso il test è stato lasciato attivo, in modo che possiate comunque avere un'idea approssimativa dei benefici prestazionali offerti dagli oggetti **Atomic**.

In **main()** il test viene eseguito più volte: il numero di ripetizioni predefinito è cinque, ma questo valore è modificabile. A ogni ripetizione il numero di cicli di test viene raddoppiato per mostrare i diversi comportamenti dei mutex quando vengono eseguiti per un numero di volte via via crescente. L'output evidenzia risultati alquanto sorprendenti: nelle prime quattro ripetizioni la parola chiave **synchronized** sembra essere più efficiente rispetto all'utilizzo di un **lock** o di **Atomic**; improvvisamente, però, si raggiunge una soglia oltre la quale **synchronized** sembra diventare inefficace, mentre all'apparenza **lock** e **Atomic** mantengono a grandi linee le stesse prestazioni del test **BaseLine**, rivelandosi quindi molto più efficienti di **synchronized**.

Tenete presente che questo programma offre soltanto un'indicazione delle differenze tra i vari approcci mutex, e l'output si riferisce alle differenze riscontrate sul computer dell'autore e in condizioni di utilizzo specifiche. Eseguendo qualche esperimento noterete che possono presentarsi variazioni rilevanti, qualora si utilizzino vari numeri di thread e il programma venga eseguito per periodi di tempo più lunghi. Ricordate che alcune ottimizzazioni a runtime non hanno luogo finché un programma non viene eseguito per diversi minuti e, nel caso dei programmi server, per parecchie ore.

Detto questo è piuttosto evidente che **lock** è generalmente più efficiente di **synchronized**, e sembra anche che gli oneri connessi a **synchronized** varino in modo considerevole, mentre i **lock** si mantengono relativamente costanti.

Questo significa forse che non dovreste mai utilizzare la parola chiave **synchronized**? Sono due i fattori da considerare: in primo luogo in **SynchronizationComparisons.java** i corpi dei metodi soggetti a mutex sono davvero ridotti. Di norma si tratta di una buona pratica: i mutex interessano unicamente le sezioni di codice in cui non è possibile adottare tecniche alternative. Nella pratica, tuttavia, le sezioni soggette a mutex potrebbero essere più grandi di quelle utilizzate nell'esempio, pertanto la percentuale di tempo trascorsa nel corpo del codice sarà probabilmente molto maggiore di quella relativa alle entrate e uscite dalla condizione di mutex, e questo potrebbe annullare il beneficio di maggiore velocità che caratterizza i mutex. Ovviamente, l'unico modo per accertarsene è provare criteri diversi e analizzarne gli effetti: tenete presente che queste verifiche dovrebbero avvenire soltanto in sede di ottimizzazione.

In secondo luogo, l'analisi degli esempi di questo capitolo evidenzia che la parola chiave **synchronized** produce codice molto più leggibile rispetto alla forma idiomatica **lock-try/finally-unlock** richiesta dai **Lock**: questo è il motivo che giustifica la netta prevalenza della parola chiave **synchronized** negli esempi. Come si è detto, il codice viene letto molto più spesso di quanto non venga scritto: quando si programma è più importante comunicare con altri esseri umani di quanto non sia comunicare con il computer, di conseguenza la leggibilità del codice è un fattore critico. Ha quindi senso iniziare a utilizzare **synchronized** e passare eventualmente agli oggetti **Lock** soltanto in fase di ottimizzazione delle prestazioni.

Per concludere, malgrado sia utile poter utilizzare le classi **Atomic** in un programma concorrente, dovete essere consapevoli del fatto che, come si è visto in **SynchronizationComparisons.java**, gli oggetti **Atomic** sono pratici soltanto in casi molto semplici, di solito quando viene modificato un solo oggetto **Atomic** e quando tale oggetto è indipendente da tutti gli altri. È più sicuro cominciare con gli approcci mutex tradizionali e poi passare alle classi **Atomic**, nel caso le prestazioni lo richiedano.

### **Contenitori non soggetti a lock**

Come avete visto nel Volume I, Capitolo II i contenitori sono uno strumento primario in programmazione, quindi anche nella programmazione concorrente. Per questo motivo i contenitori "storici" di Java, come **Vector** e **Hashtable**, disponevano di numerosi metodi **synchronized** che provocavano un sovraccarico di lavoro inaccettabile se utilizzati in applicazioni a thread



singolo. La libreria di contenitori di Java 1.2, invece, non era sincronizzata, ma forniva la classe **Collections** contenente vari metodi di decorazione **static "synchronized"** per sincronizzare i diversi tipi di contenitori. Anche se questo ha rappresentato un miglioramento, consentendo di scegliere se utilizzare la sincronizzazione con il contenitore, l'onere era ancora basato sul locking **synchronized**. Java SE5 ha aggiunto nuovi contenitori concepiti specificamente per incrementare le prestazioni in un'ottica di sicurezza dei thread, ricorrendo a tecniche "intelligenti" per eliminare i meccanismi di lock.

La strategia generica su cui si basano questi contenitori privi di lock (*lock-free*) è la seguente: le modifiche ai contenitori possono avvenire nello stesso istante in cui avviene la lettura, purché il "lettore" (parte del codice, istruzione o altro programma) possa soltanto vedere i risultati delle modifiche complete. Una modifica viene eseguita su una copia separata di una porzione della struttura di dati, o talvolta su una copia dell'intera struttura dati, e questa copia è invisibile durante l'operazione di modifica. Soltanto a modifica completa la struttura modificata viene scambiata in modo atomico con la struttura dati originaria e, a quel punto, i "lettori" vedranno la modifica.

In un oggetto **CopyOnWriteArrayList** una scrittura provoca la creazione di una copia dell'intero array sottostante. L'array originale viene mantenuto, in modo che il "lettore" possa leggerlo in modalità sicura, mentre la copia dell'array viene modificata. Quando la modifica è stata completata un'operazione atomica scambia il vecchio array con quello nuovo, affinché le successive letture abbiano accesso alle nuove informazioni. Uno dei vantaggi dell'oggetto **CopyOnWriteArrayList** è che non solleva un'eccezione **ConcurrentModificationException** quando più iteratori stanno esaminando e modificando la lista, di conseguenza non è necessario scrivere codice apposito per intercettare queste eccezioni.

**CopyOnWriteArrayList** si serve di **CopyOnWriteArrayList** per implementare il proprio comportamento lock-free.

**ConcurrentHashMap** e **ConcurrentLinkedQueue** ricorrono a tecniche simili per consentire letture e scritture simultanee; in questo caso, però, non viene copiato e modificato l'intero contenitore ma soltanto le parti necessarie. Tuttavia i lettori non rileveranno comunque alcuna modifica prima che il procedimento sia completato. **ConcurrentHashMap** non solleva eccezioni di tipo **ConcurrentModificationException**.

### Problemi di ottimizzazione

La lettura di un contenitore lock-free sarà di norma molto più rapida di quella delle relative controparti **synchronized**, poiché viene eliminato l'onere gestionale di acquisizione e rilascio dei lock. Questo è vero anche per un



numero limitato di scritture in un contenitore lock-free, tuttavia sarebbe interessante avere un'idea del significato di "numero limitato". Questi paragrafi forniranno qualche nozione approssimativa della differenza di prestazioni tra questi contenitori, in condizioni di utilizzo diverse.

Inizierete analizzando un framework generico per l'esecuzione di test su un tipo di contenitore qualunque, compreso **Map**. Il parametro generico **C** rappresenta il tipo del contenitore.

```
//: concurrency/Tester.java
// Framework per testare le prestazioni di contenitori
// concorrenti.
import java.util.concurrent.*;
import net.mindview.util.*;

public abstract class Tester<C> {
    static int testReps = 10;
    static int testCycles = 1000;
    static int containerSize = 1000;
    abstract C containerInitializer();
    abstract void startReadersAndWriters();
    C testContainer;
    String testId;
    int nReaders;
    int nWriters;
    volatile long readResult = 0;
    volatile long readTime = 0;
    volatile long writeTime = 0;
    CountDownLatch endLatch;
    static ExecutorService exec =
        Executors.newCachedThreadPool();
    Integer[] writeData;
    Tester(String testId, int nReaders, int nWriters) {
        this.testId = testId + " " +
            nReaders + "r " + nWriters + "w";
        this.nReaders = nReaders;
        this.nWriters = nWriters;
        writeData = Generated.array(Integer.class,
```



```
    new RandomGenerator.Integer(), containerSize);
for(int i = 0; i < testReps; i++) {
    runTest();
    readTime = 0;
    writeTime = 0;
}
}
void runTest() {
    endLatch = new CountDownLatch(nReaders + nWriters);
    testContainer = containerInitializer();
    startReadersAndWriters();
    try {
        endLatch.await();
    } catch(InterruptedException ex) {
        System.out.println("endLatch interrupted");
    }
    System.out.printf("%-27s %14d %14d\n",
                      testId, readTime, writeTime);
    if(readTime != 0 && writeTime != 0)
        System.out.printf("%-27s %14d\n",
                          "readTime + writeTime =", readTime + writeTime);
}
abstract class TestTask implements Runnable {
    abstract void test();
    abstract void putResults();
    long duration;
    public void run() {
        long startTime = System.nanoTime();
        test();
        duration = System.nanoTime() - startTime;
        synchronized(Tester.this) {
            putResults();
        }
        endLatch.countDown();
    }
}
public static void initMain(String[] args) {
```



```

if(args.length > 0)
    testReps = new Integer(args[0]);
if(args.length > 1)
    testCycles = new Integer(args[1]);
if(args.length > 2)
    containerSize = new Integer(args[2]);
System.out.printf("%-27s %14s %14s\n",
    "Type", "Read time", "Write time");
}
} //://:-

```

Il metodo **abstract containerInitializer()** restituisce il contenitore inizializzato da esaminare, poi memorizzato nel campo **testContainer**. L'altro metodo **abstract, startReadersAndWriters()**, avvia i task che leggono e modificano il contenitore esaminato. Vengono eseguiti diversi test con un numero variabile di lettori e scrittori al fine di valutare gli effetti del conflitto di lock, per i contenitori **synchronized**, e delle scritture, per i contenitori lock-free.

Al costruttore vengono dapprima passate varie informazioni sul test, i cui identificativi sono autoesplicativi, poi il costruttore chiama il metodo **runTest()** per un numero di volte definito da **repetitions**. Il metodo **runTest()** crea un conto alla rovescia, **CountDownLatch**, in modo che il test possa sapere quando tutti i task sono stati completati, poi inizializza il contenitore, infine chiama il metodo **startReadersAndWriters()** e attende che tutti i task siano eseguiti.

Ogni classe “Reader” o “Writer” è basata su **TestTask**, che valuta la durata del suo metodo **abstract test()**, poi chiama **putResults()** dall'interno di un blocco **synchronized** per memorizzare i risultati. Per utilizzare questa struttura di test, in cui riconoscerete il design pattern Template Method, dovete ereditare da **Tester** il particolare tipo di contenitore da esaminare e fornire classi **Reader** e **Writer** appropriate.

```

//: concurrency/ListComparisons.java
// {Args: 1 10 10} (Rapida verifica in fase di compilazione)
// Confronto approssimativo di prestazioni tra List thread-safe.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

abstract class ListTest extends Tester<List<Integer>> {
    ListTest(String testId, int nReaders, int nWriters) {

```



```
super(testId, nReaders, nWriters);
}

class Reader extends TestTask {
    long result = 0;
    void test() {
        for(long i = 0; i < testCycles; i++)
            for(int index = 0; index < containerSize; index++)
                result += testContainer.get(index);
    }
    void putResults() {
        readResult += result;
        readTime += duration;
    }
}
class Writer extends TestTask {
    void test() {
        for(long i = 0; i < testCycles; i++)
            for(int index = 0; index < containerSize; index++)
                testContainer.set(index, writeData[index]);
    }
    void putResults() {
        writeTime += duration;
    }
}
void startReadersAndWriters() {
    for(int i = 0; i < nReaders; i++)
        exec.execute(new Reader());
    for(int i = 0; i < nWriters; i++)
        exec.execute(new Writer());
}
}

class SynchronizedArrayListTest extends ListTest {
    List<Integer> containerInitializer() {
        return Collections.synchronizedList(
            new ArrayList<Integer>()
```



```
        new CountingIntegerList(containerSize)));
    }
    SynchronizedArrayListTest(int nReaders, int nWriters) {
        super("Synched ArrayList", nReaders, nWriters);
    }
}

class CopyOnWriteArrayListTest extends ListTest {
    List<Integer> containerInitializer() {
        return new CopyOnWriteArrayList<Integer>(
            new CountingIntegerList(containerSize));
    }
    CopyOnWriteArrayListTest(int nReaders, int nWriters) {
        super("CopyOnWriteArrayList", nReaders, nWriters);
    }
}

public class ListComparisons {
    public static void main(String[] args) {
        Tester.initMain(args);
        new SynchronizedArrayListTest(10, 0);
        new SynchronizedArrayListTest(9, 1);
        new SynchronizedArrayListTest(5, 5);
        new CopyOnWriteArrayListTest(10, 0);
        new CopyOnWriteArrayListTest(9, 1);
        new CopyOnWriteArrayListTest(5, 5);
        Tester.exec.shutdown();
    }
} /* Output: (Esempio)
Type          Read time      Write time
Synched ArrayList 10r 0w    232158294700      0
Synched ArrayList 9r 1w    198947618203    24918613399
readTime + writeTime =    223866231602
Synched ArrayList 5r 5w    117367305062    132176613508
readTime + writeTime =    249543918570
CopyOnWriteArrayList 10r 0w    758386889      0
CopyOnWriteArrayList 9r 1w    741305671    136145237
```



```
readTime + writeTime = 877450908
CopyOnWriteArrayList 5r 5w 212763075 67967464300
readTime + writeTime = 68180227375
*///:-
```

In `ListTest` le classi `Reader` e `Writer` eseguono le azioni specifiche per una `List<Integer>`. Nel metodo `Reader.putResults()` la durata (`duration`) viene memorizzata, come pure `result`, per impedire che i calcoli vengano continuamente ottimizzati. Viene poi definito il metodo `startReadersAndWriters()` per creare ed eseguire i `Reader` e i `Writer` specifici.

Una volta che è stato creato, `ListTest` deve essere ulteriormente ereditato per sovrascrivere il metodo `containerInitializer()`, allo scopo di generare e inizializzare i contenitori di test specifici.

In `main()` potete vedere le variazioni sui test con diverse quantità di lettori e scrittori. Tenete presente che è possibile cambiare le variabili del test fornendo parametri da riga di comando, che verranno interpretati da `Tester.initMain(args)`.

In modo predefinito il programma esegue ogni test dieci volte; questo contribuisce a stabilizzare l'output, variabile a causa delle varie attività che possono essere intraprese dalla JVM, quali le procedure di ottimizzazione e la garbage collection.<sup>21</sup>

L'output di esempio è stato modificato per mostrare solo l'ultima iterazione di ogni test. Notate che un `synchronized ArrayList` ha approssimativamente le stesse prestazioni, a prescindere dal numero di lettori e di scrittori. I lettori si contendono l'un l'altro i lock, e così fanno anche gli scrittori. Il contenitore `CopyOnWriteArrayList`, tuttavia, è nettamente più veloce in assenza di scritture e rimane ancora molto più veloce con cinque scritture. Sembra dunque possibile una ragionevole discrezionalità nell'utilizzo di `CopyOnWriteArrayList`; l'impatto della scrittura non sembra eccedere l'effetto della sincronizzazione dell'intera lista, per un certo periodo. Naturalmente dovrete provare i due diversi approcci nella vostra applicazione per sapere con certezza quale sia il migliore.

Ricordate che questo non è un buon benchmark come riferimento di valori assoluti; i valori che otterrete dall'esecuzione di questo programma saranno certamente differenti. L'obiettivo di questo esempio è soltanto quello di darvi un'idea del comportamento dei due tipi di contenitori.

21. Per un'introduzione al benchmarking sotto l'influenza della compilazione dinamica di Java, consultate la documentazione all'indirizzo [www-128.ibm.com/developerworks/library/j-jtp12214](http://www-128.ibm.com/developerworks/library/j-jtp12214).



Poiché **CopyOnWriteArraySet** utilizza **CopyOnWriteArrayList** il suo comportamento sarà simile, pertanto non è necessario prevedere un test apposito.

### Confronto tra implementazioni Map

Potete utilizzare lo stesso framework per ottenere una panoramica generale delle prestazioni di una **synchronized HashMap**, confrontandola con una **ConcurrentHashMap**.

```
//: concurrency/MapComparisons.java
// {Args: 1 10 10} (Rapida verifica in fase di compilazione)
// Confronto approssimativo di prestazioni tra Map
// thread-safe.

import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

abstract class MapTest
extends Tester<Map<Integer, Integer>> {
    MapTest(String testId, int nReaders, int nWriters) {
        super(testId, nReaders, nWriters);
    }
    class Reader extends TestTask {
        long result = 0;
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
                    result += testContainer.get(index);
        }
        void putResults() {
            readResult += result;
            readTime += duration;
        }
    }
    class Writer extends TestTask {
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
```



```
        testContainer.put(index, writeData[index]);
    }
    void putResults() {
        writeTime += duration;
    }
}
void startReadersAndWriters() {
    for(int i = 0; i < nReaders; i++)
        exec.execute(new Reader());
    for(int i = 0; i < nWriters; i++)
        exec.execute(new Writer());
}
}

class SynchronizedHashMapTest extends MapTest {
    Map<Integer, Integer> containerInitializer() {
        return Collections.synchronizedMap(
            new HashMap<Integer, Integer>(
                MapData.map(
                    new CountingGenerator.Integer(),
                    new CountingGenerator.Integer(),
                    containerSize)));
    }
    SynchronizedHashMapTest(int nReaders, int nWriters) {
        super("Synched HashMap", nReaders, nWriters);
    }
}

class ConcurrentHashMapTest extends MapTest {
    Map<Integer, Integer> containerInitializer() {
        return new ConcurrentHashMap<Integer, Integer>(
            MapData.map(
                new CountingGenerator.Integer(),
                new CountingGenerator.Integer(), containerSize));
    }
    ConcurrentHashMapTest(int nReaders, int nWriters) {
```



```

        super("ConcurrentHashMap", nReaders, nWriters);
    }
}

public class MapComparisons {
    public static void main(String[] args) {
        Tester.initMain(args);
        new SynchronizedHashMapTest(10, 0);
        new SynchronizedHashMapTest(9, 1);
        new SynchronizedHashMapTest(5, 5);
        new ConcurrentHashMapTest(10, 0);
        new ConcurrentHashMapTest(9, 1);
        new ConcurrentHashMapTest(5, 5);
        Tester.exec.shutdown();
    }
} /* Output: (Esempio)
Type           Read time      Write time
Synched HashMap 10r 0w      306052025049          0
Synched HashMap 9r 1w      428319156207      47697347568
readTime + writeTime =      476016503775
Synched HashMap 5r 5w      243956877760      244012003202
readTime + writeTime =      487968880962
ConcurrentHashMap 10r 0w     23352654318          0
ConcurrentHashMap 9r 1w     18833089400      1541853224
readTime + writeTime =      20374942624
ConcurrentHashMap 5r 5w      12037625732      11850489099
readTime + writeTime =      23888114831
*///:~
```

L'impatto dovuto all'aggiunta di scrittori a una **ConcurrentHashMap** è ancor meno evidente di quello su un **CopyOnWriteArrayList**, ma **ConcurrentHashMap** si serve di una tecnica diversa, che evidentemente riduce al minimo l'effetto delle scritture.

### **Lock ottimistico**

Gli oggetti **Atomic** eseguono operazioni atomiche come **decrementAndGet()**, tuttavia alcune classi **Atomic** permettono anche il cosiddetto *lock ottimistico* (*optimistic locking*). In termini Java, questa espressione significa che al mo-



mento di eseguire un calcolo non viene realmente utilizzato un mutex, ma quando il calcolo è terminato e viene aggiornato l'oggetto **Atomic**, si utilizza un metodo chiamato **compareAndSet()**. A questo metodo vengono passati il vecchio e il nuovo valore: se il vecchio valore non concorda con quello presente nell'oggetto **Atomic** l'operazione non riesce, poiché evidentemente qualche altro task, nel frattempo, ha modificato l'oggetto. Ricordate che in una situazione analoga di norma utilizzereste un mutex (**synchronized** o **Lock**) per impedire la modifica di un oggetto da parte di più task contemporaneamente; in questo caso, invece, siete "ottimisti" e rinunciate a bloccare i dati, sperando che nessun altro task si faccia avanti e lo modifichi. Anche qui tutto ha senso in un'ottica di prestazioni, poiché ricorrendo a un oggetto **Atomic** anziché a **synchronized** o **Lock**, potreste ottenere benefici prestazionali notevoli.

Ma che cosa succede se l'operazione **compareAndSet()** fallisce? Se il metodo **compareAndSet()** non ha successo, dovrete decidere che cosa fare: questo è molto importante poiché se non potete fare altro che recuperare dall'errore, non sarà consentito utilizzare questa tecnica ma dovete ricorrere ai normali mutex. Potreste pensare di eseguire nuovamente l'operazione, e forse questa andrà bene la seconda volta. Oppure, probabilmente è meglio ignorare il problema: in alcune simulazioni, quando va perso un punto di riferimento, alla fine esso può essere ricostruito in un'ottica più generale. Naturalmente dovete comprendere a fondo il vostro modello per sapere se questo è fattibile.

Considerate una simulazione fittizia che consiste di 100.000 "geni" di lunghezza 30, un possibile inizio di qualche algoritmo genetico. Supponete che ogni "evoluzione" di questo algoritmo richieda calcoli molto onerosi, pertanto decidete di utilizzare un sistema multiprocessore per distribuire i task e migliorare le prestazioni. In aggiunta scegliete di utilizzare oggetti **Atomic** anziché oggetti **Lock** per eliminare l'onere gestionale legato ai mutex.

Naturalmente avrete optato per questa soluzione solo dopo avere scritto una prima versione del codice, semplice e funzionante, utilizzando la parola chiave **synchronized**: dopo avere eseguito il programma ed esservi resi conto che è troppo lento, avete iniziato a servirvi di tecniche volte a migliorarne le prestazioni. A causa della natura del vostro modello, se si verifica una collisione in fase di calcolo il task che rileva il conflitto può semplicemente ignorarlo e deciderà di non aggiornare il relativo valore. Ecco un esempio di questo tipo di simulazione.

```
//: concurrency/FastSimulation.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
```



```
import java.util.*;
import static net.mindview.util.Print.*;

public class FastSimulation {
    static final int N_ELEMENTS = 100000;
    static final int N_GENES = 30;
    static final int N_EVOLVERS = 50;
    static final AtomicInteger[][] GRID =
        new AtomicInteger[N_ELEMENTS][N_GENES];
    static Random rand = new Random(47);
    static class Evolver implements Runnable {
        public void run() {
            while(!Thread.interrupted()) {
                // Seleziona casualmente un elemento da elaborare:
                int element = rand.nextInt(N_ELEMENTS);
                for(int i = 0; i < N_GENES; i++) {
                    int previous = element - 1;
                    if(previous < 0) previous = N_ELEMENTS - 1;
                    int next = element + 1;
                    if(next >= N_ELEMENTS) next = 0;
                    int oldvalue = GRID[element][i].get();
                    // Esegue alcuni tipi di calcolo di modellazione:
                    int newvalue = oldvalue +
                        GRID[previous][i].get() + GRID[next][i].get();
                    newvalue /= 3; // Calcola la media dei tre valori
                    if(!GRID[element][i]
                        .compareAndSet(oldvalue, newvalue)) {
                        // Qui impostate i criteri per gestire i
                        // malfunzionamenti. In questo caso il problema
                        // viene soltanto segnalato e ignorato; alla fine
                        // se ne occupera' il modello.
                        print("Old value changed from " + oldvalue);
                    }
                }
            }
        }
    }
}
```



```
public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < N_ELEMENTS; i++)
        for(int j = 0; j < N_GENES; j++)
            GRID[i][j] = new AtomicInteger(rand.nextInt(1000));
    for(int i = 0; i < N_EVOLVERS; i++)
        exec.execute(new Evolver());
    TimeUnit.SECONDS.sleep(5);
    exec.shutdownNow();
}
} /* (Da eseguire per visualizzare l'output) *///:-
```

Gli elementi sono disposti all'interno di un array, supponendo che la scelta di adottare un array contribuisca migliorare le prestazioni: questo presupposto sarà l'oggetto del prossimo esercizio. Ogni oggetto **Evolver** calcola la media tra il proprio valore, quello precedente e quello successivo; in caso di errore durante l'aggiornamento, **Evolver** si limita a visualizzare il valore e prosegue. Notate che in questo programma non sono stati utilizzati mutex.

**Esercizio 39 (6) FastSimulation.java** parte da presupposti ragionevoli? Provate a utilizzare un array di normali **int** invece dei valori **AtomicInteger** e utilizzate mutex **Lock**. Confrontate le prestazioni delle due versioni del programma.

### ***Lock di tipo ReadWriteLock***

**ReadWriteLock** è concepito per ottimizzare situazioni in cui scrivete relativamente di rado in una struttura dati, che però viene letta spesso da numerosi task.

Il **ReadWriteLock** consente di avere molti lettori contemporaneamente, purché nessuno tenti di scrivere. Se viene acquisito il lock in scrittura, a nessun lettore è consentito l'accesso fino al rilascio del lock.

È assolutamente aleatorio ipotizzare se un **ReadWriteLock** migliorerà le prestazioni del vostro programma: questo dipende da vari fattori, per esempio la frequenza di lettura dei dati confrontata con la frequenza di modifica, i tempi delle operazioni di lettura e scrittura (il lock è più complesso, quindi operazioni brevi non rileveranno benefici), il numero di conflitti tra i thread, e il fatto che il programma venga eseguito su sistema multiprocessore oppure monoprocesso. In pratica, il solo modo per sapere se il vostro programma



potrà trarre vantaggio da **ReadWriteLock** è provarlo. L'esempio seguente mostra soltanto l'utilizzo più elementare di **ReadWriteLock**.

```
//: concurrency/ReaderWriterList.java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ReaderWriterList<T> {
    private ArrayList<T> lockedList;
    // Esegue il corretto ordinamento:
    private ReentrantReadWriteLock lock =
        new ReentrantReadWriteLock(true);
    public ReaderWriterList(int size, T initialValue) {
        lockedList = new ArrayList<T>(
            Collections.nCopies(size, initialValue));
    }
    public T set(int index, T element) {
        Lock wlock = lock.writeLock();
        wlock.lock();
        try {
            return lockedList.set(index, element);
        } finally {
            wlock.unlock();
        }
    }
    public T get(int index) {
        Lock rlock = lock.readLock();
        rlock.lock();
        try {
            // Mostra che lettori multipli
            // potrebbero acquisire il lock in lettura:
            if(lock.getReadLockCount() > 1)
                print(lock.getReadLockCount());
            return lockedList.get(index);
        } finally {
    }
```



```
    rlock.unlock();
}
}

public static void main(String[] args) throws Exception {
    new ReaderWriterListTest(30, 1);
}
}

class ReaderWriterListTest {
    ExecutorService exec = Executors.newCachedThreadPool();
    private final static int SIZE = 100;
    private static Random rand = new Random(47);
    private ReaderWriterList<Integer> list =
        new ReaderWriterList<Integer>(SIZE, 0);
    private class Writer implements Runnable {
        public void run() {
            try {
                for(int i = 0; i < 20; i++) { // Test di 2 secondi
                    list.set(i, rand.nextInt());
                    TimeUnit.MILLISECONDS.sleep(100);
                }
            } catch(InterruptedException e) {
                // Un modo accettabile per uscire
            }
            print("Writer finished, shutting down");
            exec.shutdownNow();
        }
    }
    private class Reader implements Runnable {
        public void run() {
            try {
                while(!Thread.interrupted()) {
                    for(int i = 0; i < SIZE; i++) {
                        list.get(i);
                        TimeUnit.MILLISECONDS.sleep(1);
                    }
                }
            }
        }
    }
}
```



```
        } catch(InterruptedException e) {
            // Un modo accettabile per uscire
        }
    }
}

public ReaderWriterListTest(int readers, int writers) {
    for(int i = 0; i < readers; i++)
        exec.execute(new Reader());
    for(int i = 0; i < writers; i++)
        exec.execute(new Writer());
}
} /* (Da eseguire per visualizzare l'output) *///:~
```

Un **ReaderWriterList** può contenere un numero fisso di elementi di qualsiasi tipo. Dovete fornire al costruttore la dimensione della lista da produrre e un oggetto iniziale con cui popolarla.

Il metodo **set()** acquisisce il lock in scrittura per chiamare il metodo sottostante **ArrayList.set()**, mentre il metodo **get()** acquisisce il lock in lettura per chiamare **ArrayList.get()**. Inoltre **get()** verifica se più lettori hanno acquisito il lock: in caso affermativo ne mostra il numero, per dimostrare che tale acquisizione è possibile.

Per provare la **ReaderWriterList**, **ReaderWriterListTest** crea i task di scrittura e di lettura per un **ReaderWriterList<Integer>**. Osservate che il numero delle scritture è inferiore a quello delle letture.

Consultando la documentazione del JDK per **ReentrantReadWriteLock**, noterete che questa classe, oltre a disporre di altri metodi, offre caratteristiche di "imparzialità" e "discrezionalità".

Il **ReadWriteLock** è uno strumento piuttosto specializzato, che dovreste utilizzare soltanto quando volete migliorare le prestazioni del vostro codice. Ovviamente, la prima versione del vostro programma dovrebbe utilizzare la sincronizzazione diretta, e soltanto se necessario dovreste integrare oggetti **ReadWriteLock**.

**Esercizio 40** (6) Sulla falsariga di **ReaderWriterList.java** create una **ReaderWriterMap** utilizzando una **HashMap**: valutatene le prestazioni modificando **MapComparisons.java**. Come si comporta rispetto a una **synchronized HashMap** e a una **ConcurrentHashMap**?



## Oggetti attivi

Dopo avere lavorato su questo capitolo potreste esservi fatti l'idea che il corretto utilizzo del threading in Java sia molto complesso e difficile. Inoltre, all'apparenza potrebbe sembrarvi in qualche modo controproducente: anche se i task lavorano in parallelo, il programmatore deve impegnarsi a fondo per impedire che interferiscano l'un l'altro.

I lettori che hanno esperienza con il linguaggio Assembler potrebbero ritenere che l'approccio alla scrittura di programmi concorrenti sia simile: ogni dettaglio è importante, siete responsabili di tutto e non disponete di "reti di sicurezza", sotto forma di controlli offerti dal compilatore.

Esiste forse un problema nel modello di threading in sé? Dopotutto tale modello deriva, quasi del tutto invariato, dal mondo della programmazione procedurale. Potrebbe esistere un modello di concorrenza diverso, più adatto alla programmazione orientata agli oggetti?

Un approccio alternativo è costituito dai cosiddetti *active objects* (oggetti attivi) o *actors* (attori).<sup>22</sup>

Gli oggetti sono chiamati "attivi" poiché ciascuno di essi mantiene un suo thread di lavoro e una sua coda di messaggi e tutte le richieste a questo oggetto vengono accodate per essere eseguite una alla volta. Quindi, con gli oggetti attivi si serializzano i messaggi invece dei metodi: questo significa che non dovete più cauterlarvi dal tipico problema del task che si interrompe a metà del suo ciclo.

Quando viene inviato a un oggetto attivo, il messaggio è trasformato in un task che si inserisce nella coda dell'oggetto per essere eseguito in un momento successivo. L'oggetto **Future** di Java SE5 risulta assai pratico per implementare questo schema. Il semplice esempio che segue dispone di due metodi che accodano le chiamate di metodo.

```
//: concurrency/ActiveObjectDemo.java
// Ai metodi asincroni potete passare come argomenti solo
// valori costanti, valori immutabili, "oggetti disconnessi"
// o altri oggetti attivi.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

22. L'autore ringrazia Allen Holub per avergli dedicato tanto tempo per illustrargli questi concetti.



```
public class ActiveObjectDemo {  
    private ExecutorService ex =  
        Executors.newSingleThreadExecutor();  
    private Random rand = new Random(47);  
    // Inserisce un ritardo casuale per creare  
    // l'illusione del tempo di calcolo:  
    private void pause(int factor) {  
        try {  
            TimeUnit.MILLISECONDS.sleep(  
                100 + rand.nextInt(factor));  
        } catch(InterruptedException e) {  
            print("sleep() interrupted");  
        }  
    }  
    public Future<Integer>  
    calculateInt(final int x, final int y) {  
        return ex.submit(new Callable<Integer>() {  
            public Integer call() {  
                print("starting " + x + " + " + y);  
                pause(500);  
                return x + y;  
            }  
        });  
    }  
    public Future<Float>  
    calculateFloat(final float x, final float y) {  
        return ex.submit(new Callable<Float>() {  
            public Float call() {  
                print("starting " + x + " + " + y);  
                pause(2000);  
                return x + y;  
            }  
        });  
    }  
    public void shutdown() { ex.shutdown(); }  
    public static void main(String[] args) {  
        ActiveObjectDemo d1 = new ActiveObjectDemo();
```



```
// Impedisce la ConcurrentModificationException:  
List<Future<?>> results =  
    new CopyOnWriteArrayList<Future<?>>();  
for(float f = 0.0f; f < 1.0f; f += 0.2f)  
    results.add(d1.calculateFloat(f, f));  
for(int i = 0; i < 5; i++)  
    results.add(d1.calculateInt(i, i));  
print("All asynch calls made");  
while(results.size() > 0) {  
    for(Future<?> f : results)  
        if(f.isDone()) {  
            try {  
                print(f.get());  
            } catch(Exception e) {  
                throw new RuntimeException(e);  
            }  
            results.remove(f);  
        }  
    }  
    d1.shutdown();  
}  
} /* Output: (85% match)  
All asynch calls made  
starting 0.0 + 0.0  
starting 0.2 + 0.2  
0.0  
starting 0.4 + 0.4  
0.4  
starting 0.6 + 0.6  
0.8  
starting 0.8 + 0.8  
1.2  
starting 0 + 0  
1.6  
starting 1 + 1  
0  
starting 2 + 2
```



```
2
starting 3 + 3
4
starting 4 + 4
6
8
*///:~
```

L’“esecutore di thread singolo” (*single thread executor*) prodotto dalla chiamata a **Executors.newSingleThreadExecutor()** mantiene la propria coda illimitata bloccante e possiede un solo thread, che preleva i task dalla coda e li porta a completamento. Tutto quello che dovete fare in **calculateInt()** e **calculateFloat()** è sottoporre (**submit()**) un nuovo oggetto **Callable** in risposta a una chiamata di metodo, convertendo di fatto le chiamate in messaggi. Il corpo del metodo è contenuto all’interno del metodo **call()** nella classe interna anonima. Notate che il valore restituito da ogni metodo dell’oggetto attivo è un **Future**, con un parametro generico che è l’effettivo tipo restituito dal metodo. In tal modo la chiamata di metodo ritorna quasi immediatamente, e il chiamante utilizza il **Future** per scoprire quando il task viene completato e ottenere così il valore di ritorno reale. Questo è quanto avviene nel caso più complesso: se la chiamata non ha valore di ritorno, il processo è facilitato.

In **main()** viene creata una **List<Future<?>>** per gestire gli oggetti **Future** restituiti dai messaggi **calculateFloat()** e **calculateInt()** trasmessi all’oggetto attivo. Questa lista viene controllata con il metodo **isDone()** alla ricerca di ogni **Future**, che viene rimosso dalla **List** quando termina; i relativi risultati vengono poi elaborati. Notate che l’utilizzo di **CopyOnWriteArrayList** evita l’esigenza di copiare la **List** per impedire il verificarsi di una **ConcurrentModificationException**.

Per impedire un’associazione involontaria fra i thread, gli argomenti passati a una chiamata di metodo dell’oggetto attivo devono essere elementi di sola lettura, altri oggetti attivi, oppure *oggetti disconnessi* (*disconnected object*, termine dell’autore), vale a dire oggetti che non hanno alcun collegamento con altri task: tenete presente che questa è una condizione difficile da implementare poiché il linguaggio non fornisce alcun supporto.

Con gli oggetti attivi:

1. ogni oggetto ha il proprio thread operativo;
2. ogni oggetto ha il controllo totale dei propri campi: questo comportamento è più rigoroso di quello offerto dalle normali classi che hanno soltanto la possibilità di controllare i propri campi;



3. le comunicazioni tra gli oggetti attivi avvengono unicamente sotto forma di messaggi;
4. tutti i messaggi scambiati tra gli oggetti attivi vengono accodati.

I risultati sono davvero convincenti. Tenuto conto che il messaggio inviato da un oggetto attivo a un altro può essere bloccato soltanto dal ritardo nell'accodamento, e poiché questo ritardo è sempre molto breve e non dipende da nessun altro oggetto, in pratica la trasmissione di un messaggio non è bloccabile: l'inconveniente più grave che possa verificarsi è un breve ritardo. Dal momento che un sistema basato su oggetti attivi comunica solo tramite messaggi, due oggetti non possono bloccarsi mentre si contendono la chiamata a un metodo di un altro oggetto. Questo implica che non può avvenire il fenomeno di stallo: decisamente un grande passo in avanti.

Il thread operativo all'interno di un oggetto attivo esegue un solo messaggio per volta, quindi non esiste rischio di conflitto di risorse e non occorre preoccuparsi di sincronizzare i metodi. La sincronizzazione avviene, certo, ma a livello di messaggio, accodando le chiamate di metodo così che possa esserne eseguita una sola per volta.

Purtroppo, senza il supporto diretto da parte del compilatore l'approccio di codifica descritto è eccessivamente confuso. Tuttavia si stanno riscontrando progressi nel campo degli oggetti attivi, più precisamente in quella che viene chiamata programmazione *agent-based*. Gli agenti sono oggetti attivi, ma gli "agent system" supportano anche in modo trasparente le reti e i diversi computer. L'autore non sarebbe sorpreso se questa nuova filosofia di programmazione fosse destinata a succedere alla programmazione OOP, poiché combina gli oggetti a una soluzione di concorrenza relativamente semplice.

Se desiderate approfondire l'argomento, su Internet potrete trovare maggiori informazioni su oggetti attivi, attori e agenti: in particolare, alcuni concetti su cui si basano gli oggetti attivi derivano dalle idee esposte da Sir Charles Antony Richard Hoare nella sua teoria di comunicazione dei processi sequenziali (CSP, *Communicating Sequential Processes*) ([www.usingcsp.com](http://www.usingcsp.com)).

**Esercizio 41** (6) Aggiungete a **ActiveObjectDemo.java** un gestore di messaggi che non abbia valore di ritorno, poi chiamatelo all'interno del metodo **main()**.

**Esercizio 42** (7) Modificate **WaxOMatic.java** in modo che implementi gli oggetti attivi.

**Progetto** Utilizzate le annotazioni e la libreria Javassist per creare un'annotazione di classe `@Active` che trasformi la vostra classe in un oggetto attivo.<sup>23</sup>

## Riepilogo

L'obiettivo di questo capitolo era fornirvi le nozioni di base sulla programmazione concorrente con i thread Java, in modo da comprendere che:

1. potete eseguire più task in modo indipendente;
2. dovete considerare tutti i possibili problemi quando questi task si interrompono;
3. i task possono interferire a vicenda su risorse condivise; il mutex (lock) è lo strumento primario utilizzato per impedire queste collisioni;
4. se non vengono progettati con attenzione, i task possono portare a una situazione di stallo.

È essenziale comprendere quando utilizzare la concorrenza e quando evitarla. I motivi principali per farvi ricorso sono:

1. gestire un certo numero di task la cui interazione permetta di utilizzare le risorse del calcolatore in modo più efficiente, inclusa la capacità di distribuire i task sulle varie CPU in un'ottica di trasparenza;
2. consentire una migliore organizzazione del codice;
3. offrire maggiore praticità di utilizzo all'utente.

Il classico esempio di bilanciamento delle risorse consiste nell'utilizzare la CPU durante le attese in fase di I/O. La migliore organizzazione del codice è particolarmente evidente nelle simulazioni. L'esempio classico di praticità di utilizzo consiste nella possibilità di monitorare lo stato di un pulsante "stop" che permetta di interrompere lunghi download.

Un beneficio aggiuntivo dei thread è che forniscono un cambio di contesto di esecuzione "leggero", dell'ordine di 100 istruzioni, a differenza del cambio di contesto dei processi più "pesante" e composto solitamente da migliaia di istruzioni. Dal momento che tutti i thread di un determinato processo condividono la stessa area di memoria, un cambio di contesto "leggero" modifica solo l'esecuzione del programma e le variabili locali; di contro, un cambio di processo o cambio di contesto "pesante" deve scambiare l'intera area di memoria.

---

23. I progetti sono proposte da utilizzare, per esempio, come pianificazioni a termine. Le soluzioni ai progetti non sono incluse nella guida delle soluzioni agli esercizi.



Gli inconvenienti principali del multithreading sono quelli elencati di seguito:

1. rallentamento del sistema mentre i thread sono in attesa di accedere alle risorse condivise;
2. onere gestionale aggiuntivo legato all'implementazione dei thread;
3. inutile complessità, se il sistema non è correttamente progettato;
4. possibilità che si verifichino condizioni tali da creare problemi quali la "fame di risorse", la corsa critica, lo stallo (punto morto) e il "punto vivo" (o livelock), in cui più thread eseguono singoli task che il sistema non è in grado di concludere;
5. incoerenze tra le diverse piattaforme. Per esempio, mentre l'autore sviluppava alcuni esempi di questo manuale ha rilevato situazioni di corsa critica che si verificano rapidamente su alcuni computer ma non su altri. Se programmate un'applicazione su sistemi in cui un evento non si manifesta, potreste incontrare problemi quando distribuirete il vostro progetto.

Per quanto riguarda i thread, una delle maggiori difficoltà si riscontra quando un task potrebbe condividere una risorsa, per esempio la memoria in un oggetto, e dovete garantire che diversi task non accedano a quella risorsa simultaneamente per leggerla o modificarla. Questa eventualità richiede un utilizzo prudente dei meccanismi di bloccaggio offerti da Java, per esempio la parola chiave **synchronized**: si tratta di strumenti essenziali, che tuttavia devono essere compresi in tutta la loro importanza ed efficacia poiché possono condurre a situazioni di stallo.

Dovete anche tenere conto di un lato "artistico" nell'applicazione dei thread. Il linguaggio Java è stato concepito per permettervi di creare tutti gli oggetti che ritenete necessari per risolvere il vostro problema di programmazione. Questo, almeno in teoria: la creazione dei milioni di oggetti necessari per un'analisi ingegneristica degli elementi finiti, per esempio, in Java sarebbe impraticabile senza ricorrere al design pattern *Flyweight*. Comunque sia, sembra esistere un limite massimo nel numero di thread generabili, che oltre una certa quantità tendono a diventare poco efficienti. Questo numero massimo è un punto critico che può essere difficile da rilevare e che spesso dipenderà dal sistema operativo e dalla JVM utilizzati: può essere inferiore a cento o dell'ordine delle migliaia. Tenuto conto che di norma si creano soltanto pochi thread, in genere questo non rappresenta un limite problematico, ma in un'ottica più generale si trasforma in un vincolo che potrebbe costringervi ad adottare uno schema di concorrenza cooperativo.

A prescindere dall'apparente semplicità nella gestione del threading che caratterizza un determinato linguaggio o una libreria particolare, considerate questa "arte" alla stregua di magia nera. Ci sarà sempre qualcosa che vi assillerà quando meno ve lo aspettate. La ragione per cui il problema della cena dei fi-



losofi è interessante è che può essere risolto in modo che la situazione di stallo avvenga raramente, dando l'impressione che tutto funzioni alla perfezione.

In generale dovreste ricorrere al threading con attenzione e parsimonia. Qua-lora i problemi con il threading diventino complessi, valutate l'opportunità di impiegare un linguaggio come *Erlang*, uno dei numerosi linguaggi funzio-nali specializzati nella gestione del threading. Potrete servirvi di questo tipo di linguaggio in porzioni del vostro programma che richiedono l'utilizzo del threading, purché ne facciate un uso continuativo e le vostre esigenze siano sufficientemente complesse da giustificare questa soluzione.

### ***Ulteriori letture***

Purtroppo vi è molta disinformazione a proposito della concorrenza e non è possibile affermare di conoscere tutto quello che occorre. Ne sa qualcosa l'autore, che pur essendosi più volte convinto di conoscere la materia, in pas-sato, è certo che dovrà aspettarsi nuove rivelazioni sull'argomento in futuro. Di seguito sono indicati alcuni testi che potrebbero esservi di aiuto.

**Java Concurrency in Practice**, di Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes e Doug Lea (Addison-Wesley, 2006). In pra-tica, tutto ciò che occorre sapere sul threading in ambiente Java.

**Concurrent Programming in Java**, seconda edizione, di Doug Lea (Addison-Wesley, 2000). Anche se questo libro è apparso molto tempo prima di Java SE5, è importante tenere presente che una buona parte del lavoro di Doug si è concretizzata nelle nuove librerie di `java.util.concurrent`, pertanto questo testo è essenziale per una comprensione completa dei problemi di concor-renza. Il libro non si limita a esaminare la concorrenza in Java, ma analizza lo stato corrente delle conoscenze, con riferimento ai vari linguaggi e tecno-logie. Anche se in alcuni punti potrà risultare poco comprensibile, merita di essere riletto parecchie volte, preferibilmente a distanza di qualche mese per avere il tempo di assimilare le informazioni. Doug è una delle poche persone al mondo che comprende realmente la concorrenza, quindi l'impegno che dedicherete a questa lettura sarà un eccellente investimento.

**The Java Language Specification, terza edizione** (Capitolo 17), di Gosling, Joy, Steele e Bracha (Addison-Wesley, 2005). Le specifiche tecniche di Sun sui thread e sui lock, disponibili anche in formato elettronico, scaricabile o consultabile online all'indirizzo <http://java.sun.com/docs/books/jls/>.

*La soluzione degli esercizi è disponibile nel documento The Thinking in Java Annotated Solution Guide, in vendita all'indirizzo [www.mindview.net](http://www.mindview.net).*



# Capitolo 2

## Interfacce grafiche (GUI)



**U**n principio di riferimento fondamentale della progettazione raccomanda di “rendere facili le cose semplici e possibili le cose difficili”.<sup>1</sup>

L'obiettivo di progettazione originale della libreria GUI (*Graphical User Interface*, interfaccia grafica utente) di Java 1.0 era consentire al programmatore di costruire un'interfaccia che fosse adatta per tutte le piattaforme. Questo obiettivo non è mai stato realizzato: al contrario, la libreria AWT (*Abstract Windowing Toolkit*) di Java 1.0 ha permesso di creare GUI che erano ugualmente mediocri su tutti i sistemi. Inoltre tale libreria era caratterizzata da limiti notevoli: non ammetteva più di quattro serie di font né consentiva di accedere agli elementi più specializzati dell'interfaccia grafica messi a disposizione dal sistema operativo. Il modello di programmazione AWT di Java 1.0 era anche complesso e non OOP. Un partecipante ai seminari dell'autore, che lavorava in Sun durante la creazione di Java, ne spiega la ragione: la libreria AWT originale era stata concepita, progettata e implementata in un solo mese. Un esempio stupefacente di rendimento, senza dubbio, ma anche una lezione esemplare che dimostra l'importanza della progettazione.

---

1. Una variazione sul tema è definita come “principio del minimo stupore”, ed essenzialmente raccomanda di non sorprendere l'utente.

La situazione è migliorata con il modello a eventi AWT di Java 1.1, che fornisce un approccio più chiaro e orientato agli oggetti, implementando JavaBeans, un modello di programmazione a componenti che consente la facile creazione di ambienti di sviluppo visuali. Java 2 (JDK 1.2) ha completato la trasformazione della vecchia libreria AWT Java 1.0, sostituendola con le JFC (*Java Foundation Classes*), il componente GUI noto come "Swing". Si tratta di un ricco insieme di oggetti JavaBean, facili da utilizzare e da comprendere, che possono essere trascinati, incollati, ma anche programmati manualmente, per creare GUI apprezzabili. Come vedete, la regola della "revisione 3" dell'industria informatica, secondo la quale un software non è accettabile prima della terza release, sembra essere confermata anche per i linguaggi di programmazione.

Questo capitolo introduce la moderna libreria Swing, che attualmente è la libreria grafica di Sun per il linguaggio Java.<sup>2</sup>

Se per qualsiasi motivo dovrete servirvi della "vecchia" libreria AWT originale, per supportare vecchio codice o a causa di limitazioni nel browser, troverete l'introduzione ad AWT nella prima edizione di questo manuale, scaricabile dal sito [www.mindview.net](http://www.mindview.net). Tenete presente che alcuni componenti AWT sono tuttora presenti in Java, e in alcune situazioni dovrete servirvene.

Ricordate che questo capitolo non è un glossario completo di tutti i componenti Swing, né di tutti i metodi delle classi descritte, bensì una semplice introduzione. La libreria Swing è molto ampia, e l'obiettivo di questo capitolo è unicamente quello di fornirvi le competenze essenziali per servirsene. Se avete bisogno di maggiori informazioni e siete disposti a eseguire qualche ricerca, Swing potrà fornirvi quello che desiderate.

L'autore dà per scontato che vi siate procurati la documentazione JDK da <http://java.sun.com> e che consulterete la documentazione delle classi **javax.swing** per esaminare i dettagli e i metodi della libreria Swing. Potete anche eseguire ricerche su Internet, ma il punto migliore per iniziare è il tutorial Swing di Sun, disponibile all'indirizzo <http://java.sun.com/docs/books/tutorialuiswing>.

Numerosi manuali, alcuni piuttosto corposi, sono dedicati solamente a Swing; dovreste servirvene se vi occorrono approfondimenti o se intendete modificare i comportamenti predefiniti di questa libreria.

Imparando a utilizzare Swing scoprirete gli aspetti illustrati di seguito:

1. Swing è un modello di programmazione decisamente migliore di quello di molti altri linguaggi e ambienti di sviluppo: questo non vuol dire

---

2. IBM ha creato SWT, una nuova libreria GUI open source per il proprio editor Eclipse ([www.eclipse.org](http://www.eclipse.org)), che può essere considerata un'alternativa a Swing. SWT sarà trattata nel proseguito del capitolo.

che sia perfetto, ma rappresenta un passo avanti nella giusta direzione. JavaBeans, esaminato nella parte finale di questo capitolo, costituisce la struttura di questa libreria.

2. Gli ambienti IDE per la programmazione visuale sono un aspetto essenziale di un ambiente di sviluppo Java completo. Mediante JavaBeans e Swing l'ambiente di sviluppo visuale scriverà il codice per voi, mentre voi non dovrete fare altro che disporre i vari componenti sulle finestre servendovi degli strumenti grafici opportuni. Questo processo accelera la creazione delle interfacce grafiche e vi permette anche di fare sperimentazione, consentendovi di provare diversi schemi di progettazione al fine di scegliere il migliore.
3. Swing è piuttosto semplice, pertanto anche se utilizzerete un software IDE invece di scrivere il codice a mano il risultato dovrebbe essere facilmente comprensibile. Questo risolve un problema notevole dei vecchi ambienti IDE, che spesso creavano codice illeggibile.

Swing contiene tutti i componenti che vi aspettereste di trovare in un'interfaccia grafica moderna: pulsanti con supporto per le immagini, elenchi ad albero e tabelle, praticamente tutto. Pur essendo una libreria imponente, è progettata per avere la complessità adatta all'operazione che state eseguendo: se qualcosa è semplice, non dovrete scrivere molto codice, ma se tentate di eseguire operazioni più complesse, il vostro codice diventerà proporzionalmente più complesso.

Ciò che apprezzerete maggiormente in Swing potrebbe essere definito come "ortogonalità d'uso", ovvero il fatto che una volta che avete chiari i concetti generali della libreria potrete applicarli in ogni occasione. Soprattutto grazie alla convenzione di nomenclatura standard, mentre l'autore stava scrivendo questi esempi riusciva a prevedere con successo i nomi dei metodi da utilizzare. Questo è certamente un marchio di garanzia di una buona progettazione delle librerie. Inoltre, di norma non avrete problemi ad aggiungere altri componenti, perché tutto funzionerà correttamente.

La navigazione tramite tastiera è automatica; potete eseguire un'applicazione Swing senza utilizzare il mouse e questo non richiede alcuna programmazione supplementare. Il supporto per lo scorrimento delle finestre non richiede alcun impegno particolare: vi basta posizionare il componente in **JScrollPane** quando lo aggiungete al modulo. L'implementazione di funzionalità quali i tooltip in generale richiede soltanto una riga di codice.

Per ragioni di portabilità Swing è scritto interamente in Java.

Swing supporta anche una funzionalità piuttosto innovativa chiamata "pluggable look & feel": questa espressione indica che l'aspetto dell'interfaccia grafica varia dinamicamente per soddisfare utenti che lavorano su piattaforme e sistemi



operativi diversi. Potete persino creare il vostro aspetto personalizzato, benché con qualche difficoltà; in proposito troverete alcuni esempi su Internet.<sup>3</sup>

Malgrado tutte queste caratteristiche positive, Swing non è per tutti, né ha risolto tutti i problemi dell'interfaccia grafica utente come i progettisti auspicavano. Al termine del capitolo saranno presentate due soluzioni alternative a Swing: la libreria sponsorizzata da IBM, SWT, sviluppata per l'editore Eclipse ma disponibile come software autonomo open source, e lo strumento Flex di Adobe (ex Macromedia) per lo sviluppo di front end grafici lato client per le applicazioni web.

## Gli applet

Quando Java è stato presentato per la prima volta, gran parte dell'interesse suscitato da questo linguaggio era da imputare agli *applet*, programmi che possono essere trasmessi via Internet ed eseguiti in un browser web, all'interno di opportune *sandbox*, per ragioni di sicurezza. All'epoca molti ritenevano che gli applet Java rappresentassero la fase successiva dello sviluppo di Internet e molti dei primi manuali su Java hanno lasciato intuire che un buon motivo per interessarsi al linguaggio fosse appunto la possibilità di scrivere applet.

Per vari motivi questa rivoluzione non è mai avvenuta. Il problema predominante era che la maggior parte dei computer non disponeva del software Java necessario per eseguire gli applet; del resto, scaricare e installare un pacchetto di 10 MB per eseguire qualcosa che, forse, si sarebbe incontrato sul web era qualcosa che ben pochi utenti erano disposti a fare. Molti utenti erano persino spaventati dall'idea. L'applet Java inteso come strumento di trasmissione dell'applicazione lato client non ha mai raggiunto il grande pubblico: anche se avrete occasione di incontrare applet, di norma questi componenti sono ormai relegati a un ruolo secondario.

Questo non vuol dire che gli applet non siano una tecnologia interessante e di grande valore. Se siete in grado di garantire che gli utenti abbiano JRE installato, come accade per esempio in ambiente aziendale, gli applet (o JNLP/Java web Start, descritti nel prosieguo del capitolo), potrebbero rappresentare il meccanismo ideale per distribuire software client e per aggiornare automaticamente tutti i sistemi, senza incorrere nei normali oneri legati alla distribuzione e installazione di nuovo software.

Troverete un'introduzione alla tecnologia degli applet nei supplementi online di questo manuale, all'indirizzo [www.mindview.net](http://www.mindview.net).

3. L'esempio preferito dall'autore è "Napkin" di Ken Arnold e Alex Lam S.L., scaricabile all'indirizzo <http://napkinlaf.sourceforge.net>, che conferisce alle finestre un aspetto che sembra scarabocchiato su un tovagliolo di carta stroppiato.



## Nozioni di base su Swing

La maggior parte delle applicazioni Swing viene sviluppata all'interno di un **JFrame** di base, un modulo (*form*) che crea una finestra a prescindere dal sistema operativo utilizzato. Il titolo della finestra può essere impostato mediante il costruttore **JFrame**, come in questo esempio:

```
//: gui>HelloSwing.java
import javax.swing.*;

public class HelloSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello Swing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 100);
        frame.setVisible(true);
    }
} //://:-~
```

Il metodo **setDefaultCloseOperation()** indica a **JFrame** come deve comportarsi quando l'utente chiude la finestra, premendo il pulsante o eseguendo l'operazione equivalente da tastiera. La costante **EXIT\_ON\_CLOSE** indica di chiudere definitivamente il programma. Senza questa chiamata, in modalità predefinita Java non terminerà l'applicazione: in pratica, anche se la finestra viene chiusa, il programma rimane attivo in memoria.

Il metodo **setSize()** imposta la dimensione in pixel della finestra.

Osservate l'ultima riga dell'esempio:

```
frame.setVisible(true);
```

In assenza di questa istruzione la finestra non sarà visualizzata.

Per un esempio più interessante aggiungete un oggetto **JLabel** a **JFrame**:

```
//: gui>HelloLabel.java
import javax.swing.*;
import java.util.concurrent.*;

public class HelloLabel {
    public static void main(String[] args) throws Exception {
```



```
JFrame frame = new JFrame("Hello Swing");
JLabel label = new JLabel("A label");
frame.add(label);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(300, 100);
frame.setVisible(true);
TimeUnit.SECONDS.sleep(1);
label.setText("Hey! This is Different!");
}
} //:-
```

Il testo dell'etichetta di testo **JLabel** cambia repentinamente. Benché questo possa essere considerato da qualcuno un effetto divertente e che in un piccolo programma come questo non provochi particolari problemi, non è mai una buona idea lasciare che il thread **main()** intervenga direttamente sui componenti dell'interfaccia grafica. Swing possiede il proprio thread dedicato alla ricezione degli eventi dell'interfaccia grafica e all'aggiornamento dello schermo. Se gestite la visualizzazione con altri thread, potreste incorrere nei conflitti e nei rischi di stallo descritti nel Capitolo 1.

Di norma, altri thread, come **main()** in questo caso specifico, dovrebbero sottoporre i task da eseguire al *thread d'invio degli eventi (event dispatch thread)* di Swing.<sup>4</sup>

Per fare questo dovete inviare un task a **SwingUtilities.invokeLater()**, che lo inserirà nella *coda di eventi (event queue)*; poi il task verrà eseguito dal thread di invio degli eventi. Applicando questa procedura all'esempio precedente otterrete:

```
//: gui/SubmitLabelManipulationTask.java
import javax.swing.*;
import java.util.concurrent.*;

public class SubmitLabelManipulationTask {
    public static void main(String[] args) throws Exception {
        JFrame frame = new JFrame("Hello Swing");
        final JLabel label = new JLabel("A label");
        frame.add(label);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

4. Tecnicamente il thread per l'invio degli eventi deriva dalla libreria AWT.



```
frame.setSize(300, 100);
frame.setVisible(true);
TimeUnit.SECONDS.sleep(1);
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        label.setText("Hey! This is Different!");
    }
});
}
} //://:-
```

Ora non state gestendo direttamente **JLabel** ma inviate un oggetto **Runnable**, e il thread di invio degli eventi si incaricherà di eseguire la manipolazione quando otterrà il task nella coda di eventi. Quando eseguirà questo **Runnable** Swing non farà nient'altro, pertanto non avrete alcun problema se tutto il codice del vostro programma adotta questa tecnica di invio delle richieste a **SwingUtilities.invokeLater()**. Questo include l'avvio stesso del programma: **main()** non dovrebbe chiamare i metodi Swing come avviene nell'esempio, bensì sottoporre un task alla coda di eventi.<sup>5</sup>

Pertanto, la forma corretta del programma sarà simile alla seguente:

```
//: gui/SubmitSwingProgram.java
import javax.swing.*;
import java.util.concurrent.*;

public class SubmitSwingProgram extends JFrame {
    JLabel label;
    public SubmitSwingProgram() {
        super("Hello Swing");
        label = new JLabel("A label");
        add(label);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 100);
        setVisible(true);
    }
}
```

---

5. Questa tecnica è stata aggiunta in Java SE5, di conseguenza troverete moltissimi programmi che non sono conformi a queste direttive. Questo non significa che gli autori di tali programmi non sono preparati: sono le tecniche che si evolvono continuamente.



```
static SubmitSwingProgram ssp;
public static void main(String[] args) throws Exception {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() { ssp = new SubmitSwingProgram(); }
    });
    TimeUnit.SECONDS.sleep(1);
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            ssp.label.setText("Hey! This is Different!");
        }
    });
}
} // :~
```

Notate che la chiamata a `sleep()` non è all'interno del costruttore. Inserendola all'esterno, il testo *originale* di `JLabel` non sarà visualizzato perché il costruttore non completerà la sua esecuzione fino al termine di `sleep()`, e verrà così utilizzata la *nuova* etichetta. Invece, se `sleep()` fosse all'interno del costruttore o di qualsiasi operazione dell'interfaccia utente, bloccherebbe il thread d'invio degli eventi, il che generalmente è sintomo di cattiva programmazione.

**Esercizio 1** (1) Modificate `HelloSwing.java` per dimostrare che l'applicazione non si chiude in assenza della chiamata a `setDefaultCloseOperation()`.

**Esercizio 2** (2) Modificate `HelloLabel.java` aggiungendo un numero casuale di etichette, per dimostrare che questa operazione è di tipo dinamico.

### ***Un framework di visualizzazione***

Per combinare i concetti illustrati e ridurre il codice ridondante potete creare un framework di visualizzazione da utilizzare negli esempi di Swing nel resto del capitolo:

```
/: net/mindview/util/SwingConsole.java
// Strumento per eseguire da console programmi dimostrativi
// Swing, sia applet sia JFrame.
package net.mindview.util;
import javax.swing.*;
```



```

public class SwingConsole {
    public static void
        run(final JFrame f, final int width, final int height) {
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    f.setTitle(f.getClass().getSimpleName());
                    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    f.setSize(width, height);
                    f.setVisible(true);
                }
            });
        }
    } //:-_
}

```

Questa utility è stata inclusa nella libreria **net.mindview.util** poiché potrebbe esservi utile anche nella vostra normale attività di programmatore. Per utilizzarla, è necessario che l'applicazione sia inserita in un **JFrame**, come tutti gli esempi di questo manuale. Il metodo **static run()** imposta semplicemente il titolo della finestra al nome della classe **JFrame**.

**Esercizio 3 (3)** Modificate **SubmitSwingProgram.java** per servirvi di **SwingConsole**.

## Costruzione di un pulsante

Creare un pulsante è un'operazione semplicissima: chiamate il costruttore **JButton** passandogli l'etichetta di testo che volete sia visualizzata sul pulsante. Vedrete nel prosieguo del capitolo come eseguire operazioni più elaborate, quali l'inserimento di immagini che appariranno sui pulsanti.

**JButton** è un componente che possiede la propria piccola icona rappresentativa e che viene automaticamente ridisegnato in sede di aggiornamento della schermata; questo significa che non occorre disegnare il pulsante (o qualsiasi altro tipo di controllo grafico), ma basta disporlo nella finestra del modulo e lasciare che si disegni automaticamente. Di norma il pulsante viene inserito su un modulo del costruttore:

```

//: gui/Button1.java
// Come inserire dei pulsanti nelle applicazioni Swing.
import javax.swing.*;

```



```
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Button1 extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    public Button1() {
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
    }
    public static void main(String[] args) {
        run(new Button1(), 200, 100);
    }
} //:-~
```

Questo esempio evidenzia qualcosa di nuovo: prima che un elemento qualsiasi sia disposto sul **JFrame** viene assegnato un “gestore di layout” (in inglese, *layout manager*), di tipo **FlowLayout**. Il layout manager è lo strumento mediante il quale il pannello (*pane*) stabilisce la disposizione dei comandi in un form. In modalità predefinita **JFrame** utilizza **BorderLayout**: in questo caso tuttavia tale layout non funzionerà perché, come vedrete in seguito, ogni controllo viene coperto da ogni nuovo controllo aggiunto. **FlowLayout**, invece, fa in modo che i controlli si dispongano sul form progressivamente da sinistra a destra e dal basso verso l’alto.

**Esercizio 4 (1)** Verificate che, senza la chiamata al metodo **setLayout()**, **Button1.java** visualizzerà un solo pulsante.

## Intercettare un evento

Se compilate ed eseguite il programma precedente, quando premerete i pulsanti non accadrà alcunché. Questo è il passo successivo: scrivere il codice per specificare che cosa deve accadere. Il principio di base della programmazione *event-driven*, attraverso la quale vengono gestite le interfacce grafiche utente, consiste nel collegare eventi al codice, che risponderà a tali eventi.

In Swing questo abbinamento è realizzato in modo nitido, separando l’interfaccia, vale a dire i componenti grafici, dall’implementazione, ovvero il co-



dice da eseguire in risposta a un evento verificatosi in un componente. Ogni componente Swing può segnalare tutti gli eventi che potrebbero accadere, come pure segnalare individualmente ogni genere di evento; di conseguenza, se non vi interessasse sapere, per esempio, se il mouse si sta spostando sopra il vostro pulsante, vi basterebbe non "registrare il vostro interesse" per quell'evento. Questo è un modo molto diretto ed elegante di gestire la programmazione a eventi; una volta compresi i concetti di base potete utilizzare facilmente anche i componenti di Swing che non avete mai visto. Come vedrete in seguito, infatti, questo modello si estende a qualsiasi elemento sia classificabile come JavaBean.

Inizialmente concentrerete la vostra attenzione sugli eventi principali dei componenti di cui vi servirete. Nel caso di un **JButton**, questo "evento di interesse" è rappresentato dalla pressione del pulsante: per "registrare il vostro interesse" a questo evento chiamate il metodo **addActionListener()** di **JButton**. L'argomento da fornire al metodo è un oggetto che implementa l'interfaccia **ActionListener**, che contiene un solo metodo chiamato **actionPerformed()**. Quindi, per collegare il codice a un **JButton**, implementate l'interfaccia **ActionListener** in una classe e registrate un oggetto di quella classe con il pulsante **JButton**, mediante il metodo **addActionListener()**; la pressione del pulsante chiamerà poi il metodo **actionPerformed()**. Questo comportamento viene normalmente chiamato *callback*.

Ma quale dovrebbe essere il risultato della pressione del pulsante? Supponendo di voler vedere qualcosa che si modifica sullo schermo, introdurrete un nuovo elemento di Swing: la casella di testo **JTextField**. Si tratta di un componente nel quale l'utente può digitare direttamente testo, oppure, come in questo caso, lasciare che sia il programma a inserirlo. Esistono varie tecniche per creare un **JTextField**, la più semplice delle quali consiste nell'indicare al costruttore le dimensioni da assegnare alla casella di testo. Dopo avere posizionato **JTextField** sul form potete modificarne il contenuto con il metodo **setText()**; ricordate che **JTextField** supporta molti altri metodi, che troverete dettagliati nella documentazione JDK all'indirizzo <http://java.sun.com>. Un esempio è il seguente:

```
//: gui/Button2.java
// Come rispondere alla pressione dei pulsanti.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;
```



```
public class Button2 extends JFrame {  
    private JButton  
        b1 = new JButton("Button 1"),  
        b2 = new JButton("Button 2");  
    private JTextField txt = new JTextField(10);  
    class ButtonListener implements ActionListener {  
        public void actionPerformed(ActionEvent e) {  
            String name = ((JButton)e.getSource()).getText();  
            txt.setText(name);  
        }  
    }  
    private ButtonListener b1 = new ButtonListener();  
    public Button2() {  
        b1.addActionListener(b1);  
        b2.addActionListener(b1);  
        setLayout(new FlowLayout());  
        add(b1);  
        add(b2);  
        add(txt);  
    }  
    public static void main(String[] args) {  
        run(new Button2(), 200, 150);  
    }  
}
```

La creazione di un **JTextField** e la sua disposizione sul modulo richiedono le stesse operazioni previste per **JButton**, e valide per qualsiasi componente Swing. La novità di questo programma è la creazione della classe **ButtonListener**, che estende **ActionListener**. L'argomento del metodo **actionPerformed()**, che contiene tutte le informazioni sull'evento e la sua provenienza, è di tipo **ActionEvent**. In questo caso si è voluto descrivere la pressione del pulsante: il metodo **getSource()** restituisce l'oggetto da cui origina l'evento; poi si presume, utilizzando un cast, che l'oggetto sia un pulsante **JButton**. Il metodo **getText()** restituisce il testo presente sul pulsante; tale testo viene poi inserito nella casella **JTextField** per dimostrare che il codice è stato effettivamente attivato dalla pressione del pulsante.

Il costruttore si serve di **addActionListener()** per registrare l'oggetto **ButtonListener** con entrambi i pulsanti.



Spesso può essere più pratico codificare **ActionListener** sotto forma di classe interna anonima, soprattutto perché in genere si utilizza una sola istanza di ogni classe listener. Come vedete nell'esempio seguente, **Button2.java** può essere modificato per utilizzare una classe interna anonima:

```
//: gui/Button2b.java
// Utilizzo di classi interne anonime.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;
public class Button2b extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    private JTextField txt = new JTextField(10);
    private ActionListener bl = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String name = ((JButton)e.getSource()).getText();
            txt.setText(name);
        }
    };
    public Button2b() {
        b1.addActionListener(bl);
        b2.addActionListener(bl);
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(txt);
    }
    public static void main(String[] args) {
        run(new Button2b(), 200, 150);
    }
} ///:~
```

Gli esempi presentati nel corso di questo capitolo, quando possibile, adotteranno le classi interne.



**Esercizio 5 (4)** Create un'applicazione servendosi della classe **JTextArea**. Includetevi una casella di testo e tre pulsanti. Fate in modo che la pressione di ciascun pulsante nella casella di testo visualizzi un testo diverso.

## Area di testo

Un componente **JTextArea** è simile a una casella **JTextField** ma può contenere più righe e offre maggiori funzionalità. Un metodo molto pratico è **append()**, che vi permette di riversare l'output in una **JTextArea**. La possibilità di fare scorrere il testo della casella verso l'alto costituisce un notevole miglioramento rispetto ai programmi da riga di comando, che visualizzano i risultati su standard output. Per esempio, il programma seguente riempie un'area di testo **JTextArea** con l'output ottenuto dal generatore **Countries** presentato nel Volume 2, Capitolo 5:

```
//: gui/TextArea.java
// Utilizzo del controllo JTextArea.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

public class TextArea extends JFrame {
    private JButton
        b = new JButton("Add data"),
        c = new JButton("Clear data");
    private JTextArea t = new JTextArea(20, 40);
    private Map<String, String> m =
        new HashMap<String, String>();
    public TextArea() {
        // Usa tutti i dati:
        m.putAll(Countries.capitals());
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
```



```
        for(Map.Entry me : m.entrySet())
            t.append(me.getKey() + ": " + me.getValue()+"\n");
    }
});
c.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        t.setText("");
    }
});
setLayout(new FlowLayout());
add(new JScrollPane(t));
add(b);
add(c);
}
public static void main(String[] args) {
    run(new TextArea(), 475, 425);
}
} //:-~
```

Nel costruttore il contenitore **Map** viene popolato con i nomi delle nazioni e le relative capitali. Tenete presente che, per entrambi i pulsanti, **ActionListener** viene creato e aggiunto senza definire una variabile intermedia, dal momento che non occorre fare riferimento a questo listener nel corso del programma. Il pulsante “Add data” formatta e inserisce (**append()**) tutti i dati, mentre “Clear data” utilizza **setText()** per eliminare tutto il contenuto di **JTextArea**.

Quando il controllo **JTextArea** viene aggiunto a **JFrame**, è inserito in un **JScrollPane** che gestisce lo scorrimento, nel caso in cui il testo ecceda le dimensioni effettive della casella: per usufruire delle funzionalità di scorrimento (*scrolling*) non occorre altro. Molti programmati, valutando un’analoga operazione in altri ambienti di programmazione grafici, si sono dichiarati sorpresi dalla semplicità di Swing e dall’eccellente progettazione di componenti quali **JScrollPane**.

**Esercizio 6 (7)** Trasformate l’esempio **strings/TestRegularExpression.java** in un programma interattivo Swing che permetta di inserire una stringa di input in una **JTextArea**, e un’espressione regolare in un campo **JTextField**. I risultati dovranno essere visualizzati in un’altra casella di testo **JTextArea**.



**Esercizio 7** (5) Create un'applicazione utilizzando **SwingConsole** e aggiungetevi tutti i componenti Swing che hanno un metodo **addActionListener()**: li troverete elencati nella documentazione JDK su <http://java.sun.com>. Suggerimento: potete cercare **addActionListener()** utilizzando la funzionalità *Index*. Intercezionate poi gli eventi di tali componenti e visualizzate un messaggio appropriato per ognuno di essi all'interno di un campo di testo.

**Esercizio 8** (6) Quasi tutti i componenti Swing derivano da **Component**, che dispone di un metodo **setCursor()**. Consultate la documentazione JDK per questo metodo, poi create un'applicazione e cambiate il cursore standard con uno dei cursori alternativi presenti nella classe **Cursor**.

## Controllare la disposizione dei componenti

Avrete probabilmente notato che la tecnica per posizionare i componenti su un form Java è diversa da quella di altre interfacce grafiche. Per prima cosa tutto è presente nel codice: non esistono "risorse" che controllano la disposizione dei componenti. In secondo luogo, il modo in cui vengono disposti i componenti non è gestito sotto forma di posizioni assolute, bensì per mezzo di "layout manager" che decidono come devono essere visualizzati i componenti, in base all'ordine in cui sono stati aggiunti (**add()**) al modulo. Dimensioni, aspetto e posizione dei componenti sono notevolmente diversi da un layout manager a un altro. Inoltre i layout manager si adattano alle dimensioni dell'applet o della finestra dell'applicazione, pertanto, al variare delle dimensioni della finestra, cambieranno anche le dimensioni, l'aspetto e la posizione dei componenti.

**JApplet**, **JFrame**, **JWindow**, **JDialog**, **JPanel** ecc. possono tutti contenere e visualizzare degli oggetti **Component**. In **Container** è presente un metodo **setLayout()** che vi consente di scegliere un layout manager diverso. Nei prossimi paragrafi vedrete i vari layout manager disponibili: per alcuni il codice di esempio si limita al posizionamento di pulsanti, l'operazione indubbiamente più semplice.

### **BorderLayout**

In modalità predefinita **JFrame** utilizza il layout manager **BorderLayout**. Senza necessità di altre istruzioni, questo layout accetta qualsiasi elemento vi venga aggiunto e lo posiziona al centro, ridimensionando l'oggetto fino ai bordi.

**BorderLayout** si basa su quattro zone lungo i bordi e una zona centrale. Quando aggiungete un componente a un pannello gestito con **BorderLayout**, utilizzate il metodo **add()**, che accetta come primo argomento una costante. Tale valore può essere uno di quelli elencati di seguito.

<b>BorderLayout.NORTH</b>	Alto
<b>BorderLayout.SOUTH</b>	Basso
<b>BorderLayout.EAST</b>	Destra
<b>BorderLayout.WEST</b>	Sinistra
<b>BorderLayout.CENTER</b>	Posiziona i componenti al centro, entro i limiti di altri componenti o bordi.

Se non specificate l'area in cui posizionare l'oggetto, in modalità predefinita Java utilizzerà **CENTER**.

Questo esempio utilizza il layout manager predefinito di **JFrame**, vale a dire **BorderLayout**:

```
//: gui/BorderLayout1.java
// Dimostrazione di BorderLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class BorderLayout1 extends JFrame {
    public BorderLayout1() {
        add(BorderLayout.NORTH, new JButton("North"));
        add(BorderLayout.SOUTH, new JButton("South"));
        add(BorderLayout.EAST, new JButton("East"));
        add(BorderLayout.WEST, new JButton("West"));
        add(BorderLayout.CENTER, new JButton("Center"));
    }
    public static void main(String[] args) {
        run(new BorderLayout1(), 300, 250);
    }
} ///:-
```



Per qualsiasi disposizione diversa da **CENTER** l'elemento aggiunto viene ridimensionato in modo da adattarsi al minimo spazio lungo una dimensione, e allungato al massimo lungo l'altra; **CENTER**, invece, si espande in entrambe le dimensioni per occupare l'area centrale.

### **FlowLayout**

Questo layout dispone i componenti sul form da sinistra a destra e poi, esaurito lo spazio disponibile, scendendo di una riga e posizionando i componenti successivi di nuovo da sinistra a destra.

L'esempio seguente imposta il layout manager a **FlowLayout** e dispone dei pulsanti sul form. Noterete che con **FlowLayout** i componenti assumono le loro dimensioni "naturali": in un **JButton**, per esempio, esse corrispondono alla stringa di testo che ne costituisce l'etichetta.

```
//: gui/FlowLayout1.java
// Dimostrazione di FlowLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class FlowLayout1 extends JFrame {
    public FlowLayout1() {
        setLayout(new FlowLayout());
        for(int i = 0; i < 20; i++)
            add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        run(new FlowLayout1(), 300, 300);
    }
} ///:-
```

In un **FlowLayout** tutti i componenti verranno compattati alle loro dimensioni minime, pertanto in alcuni casi l'aspetto della finestra potrebbe sorprendervi; per esempio, poiché una **JLabel** avrà dimensioni pari alla stringa di testo che contiene, anche allineando il testo a destra l'aspetto dell'etichetta rimarrà invariato. Tenete presente che, ridimensionando la finestra, il layout manager **FlowLayout** riposiziona i componenti per adattarli alle nuove dimensioni della finestra.



### **GridLayout**

Un **GridLayout** consente di costruire una griglia in cui disporre i componenti: essi vengono posizionati nella griglia da sinistra a destra e dall'alto verso il basso. Nel costruttore dovete specificare il numero di righe e colonne che costituiscono la griglia; i controlli saranno visualizzati in proporzioni uguali.

```
//: gui/GridLayout1.java
// Dimostrazione di GridLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class GridLayout1 extends JFrame {
    public GridLayout1() {
        setLayout(new GridLayout(7,3));
        for(int i = 0; i < 20; i++)
            add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        run(new GridLayout1(), 300, 300);
    }
} ///:-
```

In questo caso sono presenti 21 caselle ma soltanto 20 pulsanti: l'ultima casella, quindi, viene lasciata vuota perché **GridLayout** non esegue alcun tipo di "bilanciamento".

### **GridBagLayout**

Il layout **GridBagLayout** vi fornisce un livello di controllo molto elevato, grazie al quale potete decidere esattamente come appariranno le varie zone della finestra e come verranno riformattate quando la finestra verrà ridimensionata. Questo è anche il layout manager più complesso, progettato in particolare per la produzione automatica del codice da parte di un generatore di interfacce grafiche: un generatore di GUI, infatti, potrebbe utilizzare il layout manager **GridBagLayout** in luogo del posizionamento assoluto dei componenti. Se il progetto è così complesso da richiedere l'utilizzo di **GridBagLayout**, dovrete servirvi di uno strumento per la generazione di inter-



facce per disporre automaticamente i componenti. Per conoscere i dettagli sul funzionamento di questo layout, fate riferimento alla guida in linea di Sun (<http://java.sun.com/docs/books/tutorial/uiswing/layout/gridbag.html>) o consultate un manuale specifico su Swing.

Come alternativa potete considerare **TableLayout**, un layout che pur non facendo parte delle librerie AWT/Swing standard è scaricabile all'indirizzo <http://java.sun.com/products/jfc/tsc/articles/tablelayout/apps/TableLayout.jar>. Questo componente si "stratifica" su **GridBagLayout**, nascondendone la complessità e facilitandone l'utilizzo. Troverete ulteriori informazioni su **TableLayout** sul sito di Sun (<http://java.sun.com/products/jfc/tsc/articles/tablelayout/>).

### **Posizionamento assoluto**

È anche possibile impostare la posizione assoluta dei componenti grafici come descritto di seguito.

1. Impostate a **null** il layout manager per il vostro **Container**, con l'istruzione **setLayout(null)**.
2. Chiamate **setBounds()** o **reshape()**, secondo la versione Java utilizzata, passando al metodo gli estremi di un riquadro di contenimento, espressi sotto forma di coordinate in pixel. Potete eseguire questa operazione nel costruttore o nel metodo **paint()**, in funzione di ciò che desiderate realizzare.

Alcuni generatori di GUI ricorrono a questo approccio, che tuttavia solitamente non è il metodo migliore per generare il codice.

### **BoxLayout**

Poiché molti programmati hanno manifestato difficoltà a comprendere **GridBagLayout** e a lavorare con questo layout manager, Swing include **BoxLayout**, che fornisce molti dei vantaggi di **GridBagLayout** senza la complessità che lo caratterizza. Vi servirete spesso di **BoxLayout** per impostare manualmente la posizione dei controlli: se il progetto diventa troppo complesso, è comunque consigliabile utilizzare un generatore di GUI.

Il layout **BoxLayout** vi consente di gestire la disposizione orizzontale e verticale dei componenti e di impostare la spaziatura tra di essi, ricorrendo a un approccio detto *struts and glue*. Potete trovare alcuni esempi di impiego del layout **BoxLayout** sul sito di Sun, all'indirizzo <http://java.sun.com/docs/books/tutorial/uiswing/layout/box.html>.



### Qual è l'approccio migliore?

Swing è una libreria potente, che vi permette di ottenere molto con poche righe di codice. Gli esempi indicati in questo libro sono piuttosto semplici e per scopi didattici ha senso scrivere il codice manualmente. In effetti potete ottenere interessanti risultati combinando tra loro semplici layout; oltre un certo livello, però, la costruzione manuale delle interfacce grafiche cessa di essere produttiva: il codice diventa troppo macchinoso e non rappresenta un buon impiego del vostro tempo di programmatore.

I progettisti di Java e Swing hanno fatto sì che il linguaggio e le librerie Java supportino gli strumenti di programmazione visuali, espressamente creati per semplificare l'attività di programmazione. Purché comprendiate che cosa avviene al di là dei layout e come gestire gli eventi che saranno descritti in seguito, non è fondamentale che conosciate in dettaglio le tecniche per posizionare manualmente i componenti. Lasciate che se ne occupi lo strumento adatto: Java, dopo tutto, è stato progettato per incrementare la produttività dei programmatori.

## Modello a eventi di Swing

Nel modello a eventi di Swing un componente può attivare (*to fire*) un evento. Ogni tipo di evento è rappresentato da una classe separata. Quando viene caricato, l'evento è ricevuto da uno o più *listener* ("ascoltatori") che agiscono su di esso: pertanto, l'origine di un evento e il punto in cui viene gestito possono essere separati. Poiché generalmente utilizzerete i componenti di Swing nella loro forma standard ma dovrete scrivere il codice personalizzato che verrà chiamato quando i componenti ricevono l'evento, questo è un ottimo esempio di segregazione tra interfaccia e implementazione.

Ogni listener di evento è un oggetto di una classe che implementa un tipo particolare di interfaccia listener. Come programmatore, tutto ciò che dovete fare è creare un oggetto listener e registrarlo con il componente che sta attivando l'evento: per fare questo chiamate un metodo **addXXXListener()** nel componente di caricamento degli eventi, in cui "XXX" rappresenta il tipo di evento. Per conoscere i tipi di evento che possono essere gestiti, osservate i nomi dei metodi "addListener": se provate a "rimanere in attesa" (*to listen*) di eventi errati, ritroverete un errore appropriato in fase di compilazione. Vedrete nel prosieguo del capitolo che anche JavaBeans utilizza i nomi dei metodi "add Listener" per determinare gli eventi gestibili per mezzo di un bean.



Tutta la logica dell'evento, quindi, andrà all'interno di una classe listener. Quando create una classe listener, l'unico limite è che deve implementare l'interfaccia adatta. Potete generare una classe listener globale, ma questa è una situazione in cui le classi interne tendono a essere abbastanza utili, non soltanto perché forniscono un raggruppamento logico delle classi listener all'interno dell'interfaccia grafica o delle classi di logica applicativa, ma anche perché un oggetto di una classe interna mantiene un riferimento al suo oggetto genitore, fornendo un meccanismo utile per eseguire le chiamate al di là dei limiti imposti dalle classi e dai sottosistemi.

Finora tutti gli esempi di questo capitolo hanno utilizzato il modello a eventi Swing, e nei prossimi paragrafi esaminerete i dettagli di questo modello.

### *Tipi di eventi e di listener*

Tutti i componenti Swing includono metodi **addXXXListener()** e **removeXXXListener()**, in modo che possiate aggiungere e rimuovere i tipi di listener adatti in ogni componente. Ricordate che "XXX" rappresenta anche l'*argomento* per il metodo, come in **addMyListener(MyListener m)**.

La tabella seguente elenca eventi, listener e metodi associati di base, con i componenti essenziali che supportano gli eventi specifici per mezzo dei metodi **addXXXListener()** e **removeXXXListener()**. Considerato che il modello di evento è stato concepito per essere estendibile, potreste trovare altri eventi e tipi di listener che non sono compresi in questa tabella.

<i>Evento, interfaccia listener e metodi add- e remove</i>	<i>Componenti che supportano l'evento</i>
ActionEvent ActionListener addActionListener() removeActionListener()	JButton, JList, JTextField, JMenuItem e i suoi derivati, compreso JCheckBoxMenuItem.
AdjustmentEvent AdjustmentListener addAdjustmentListener() removeAdjustmentListener()	JScrollbar e qualsiasi oggetto da voi creato che implementi l'interfaccia Adjustable.
ComponentEvent ComponentListener addComponentListener() removeComponentListener()	Component e i suoi derivati, inclusi JButton, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea e JTextField.



<i>Evento, interfaccia listener e metodi add- e remove</i>	<i>Componenti che supportano l'evento</i>
<b>ContainerEvent</b> <b>ContainerListener</b> addContainerListener() removeContainerListener()	<b>Container</b> e i suoi derivati, inclusi <b>JPanel</b> , <b>JApplet</b> , <b>JScrollPane</b> , <b>Window</b> , <b>JDialog</b> , <b>JFileDialog</b> e <b>JFrame</b> .
<b>FocusEvent</b> <b>FocusListener</b> addFocusListener() removeFocusListener()	<b>Component</b> e i suoi derivati.
<b>KeyEvent</b> <b>KeyListener</b> addKeyListener() removeKeyListener()	<b>Component</b> e i suoi derivati.
<b>MouseEvent</b> (clic e spostamento) <b>MouseListener</b> addMouseListener() removeMouseListener()	<b>Component</b> e i suoi derivati.
<b>MouseEvent*</b> (clic e spostamento) <b>MouseMotionListener</b> addMouseMotionListener() removeMouseMotionListener()	<b>Component</b> e i suoi derivati.
<b>WindowEvent</b> <b>WindowListener</b> addWindowListener() removeWindowListener()	<b>Window</b> e i suoi derivati, inclusi <b>JDialog</b> , <b>JFileDialog</b> e <b>JFrame</b> .
<b>ItemEvent</b> <b>ItemListener</b> addItemListener() removeItemListener()	<b>JCheckBox</b> , <b>JCheckBoxMenuItem</b> , <b>JComboBox</b> , <b>JList</b> e qualsiasi oggetto che implementi l'interfaccia <b>ItemSelectable</b> .
<b>TextEvent</b> <b>TextListener</b> addTextListener() removeTextListener()	Qualsiasi oggetto deriva da <b>JTextComponent</b> , inclusi <b>JTextArea</b> e <b>JTextField</b> .
(*) Non esiste <b>MouseMotionEvent</b> , anche se dovrebbe esserci. Le azioni di clic e spostamento sono combinate in <b>MouseEvent</b> , pertanto questa seconda occorrenza di <b>MouseEvent</b> nella tabella non è un errore.	

Come potete vedere, ogni tipo di componente supporta solo determinati tipi di eventi. L'analisi dettagliata di tutti gli eventi supportati dai vari compo-



nenti è un'attività lunga ed estenuante: una tecnica più semplice consiste nel modificare il programma **ShowMethods.java** del Volume 2, Capitolo 2, in modo che visualizzi tutti i listener di eventi supportati da qualsiasi componente Swing specifiche.

Nel Volume 2, il Capitolo 2 ha presentato la *riflessione* e utilizzato questa funzionalità per analizzare i metodi di una determinata classe: l'elenco completo dei metodi o un sottoinsieme di quelli i cui nomi corrispondano alla parola chiave fornita.

L'aspetto sorprendente della riflessione è che può mostrare automaticamente *tutti* i metodi di una classe senza costringervi a percorrere la gerarchia ereditaria, esaminando le classi di base di ogni livello. Rappresenta quindi uno strumento di programmazione eccellente, grazie al quale potrete risparmiare tempo; dal momento che i nomi della maggior parte dei metodi Java sono verbosi e descrittivi, potete cercare quelli che contengono una stringa particolare: quando avrete trovato ciò che vi occorre, non dovrete fare altro che controllare la documentazione JDK.

Di seguito è mostrata la versione grafica di **ShowMethods.java**, specializzata per ricercare i metodi "addListener" nei componenti Swing.

```
//: gui>ShowAddListeners.java
// Visualizza i metodi "addListener" di qualsiasi classe
// Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.SwingConsole.*;

public class ShowAddListeners extends JFrame {
    private JTextField name = new JTextField(25);
    private JTextArea results = new JTextArea(40, 65);
    private static Pattern addListener =
        Pattern.compile("(add\\w+?Listener\\\\(.+?\\\\))");
    private static Pattern qualifier =
        Pattern.compile("\\w+\\.");
    class Namel implements ActionListener {
        public void actionPerformed(ActionEvent e) {
```



```
String nm = name.getText().trim();
if(nm.length() == 0) {
    results.setText("No match");
    return;
}
Class<?> kind;
try {
    kind = Class.forName("javax.swing." + nm);
} catch(ClassNotFoundException ex) {
    results.setText("No match");
    return;
}
Method[] methods = kind.getMethods();
results.setText("");
for(Method m : methods) {
    Matcher matcher =
        addListener.matcher(m.toString());
    if(matcher.find())
        results.append(qualifier.matcher(
            matcher.group(1)).replaceAll("") + "\n");
}
}
}
}
public ShowAddListeners() {
    NameL nameListener = new NameL();
    name.addActionListener(nameListener);
    JPanel top = new JPanel();
    top.add(new JLabel("Swing class name (press Enter):"));
    top.add(name);
    add(BorderLayout.NORTH, top);
    add(new JScrollPane(results));
    // Dati e test iniziali:
    name.setText("JTextArea");
    nameListener.actionPerformed(
        new ActionEvent("", 0 , ""));
}
```



```
public static void main(String[] args) {
    run(new ShowAddListeners(), 600, 400);
}
} //:~
```

In nome **JTextField** digitate il nome della classe Swing da recuperare: i risultati vengono estratti utilizzando le espressioni regolari e visualizzati in **JTextArea**. Noterete che non esistono pulsanti né altri componenti per attivare l'inizio della ricerca, perché il campo **JTextField** viene controllato da un **ActionListener**: ogni volta che eseguite una modifica e premete Invio, l'elenco si aggiorna immediatamente. Se il campo di testo non è vuoto, il suo valore viene usato all'interno di **Class.forName()** per la ricerca della classe. Se il nome non è corretto, **Class.forName()** andrà in errore sollevando un'eccezione, che verrà bloccata; a quel punto la **JTextArea** verrà impostata a "No match". Invece, se digitate un nome corretto, tenendo conto anche di maiuscole e minuscole, **Class.forName()** avrà successo e **getMethods()** restituirà un array di oggetti **Method**.

Il codice si serve di due espressioni regolari. La prima, **addListener**, cerca la stringa "add" seguita da un certo numero di caratteri, poi da "Listener" e dall'elenco degli argomenti tra parentesi. Osservate che questa espressione è circondata da parentesi non sottoposte a escape, per fare in modo che sia accessibile come "gruppo" di espressione regolare al momento della corrispondenza. All'interno del metodo **NameL.ActionPerformed()** viene creato un **Matcher** passando ogni oggetto **Method** al metodo **Pattern.matcher()**. Quando viene chiamato **find()** per questo oggetto **Matcher**, esso restituisce **true** soltanto in caso di corrispondenza, nel qual caso potrete selezionare il primo gruppo di corrispondenza in parentesi, chiamando **group(1)**. Questa stringa contiene ancora i qualificatori, pertanto per toglierli viene utilizzato l'oggetto **qualifier Pattern**, come in **ShowMethods.java**.

Alla fine del costruttore, in **name** viene posto un valore iniziale e l'evento di azione è eseguito per fornire un test con i dati iniziali.

Questo programma è pratico per studiare le possibilità di un componente Swing. Una volta che conoscete quali eventi sono supportati da un determinato componente, non vi occorre altro per reagire a quell'evento. Dovrete semplicemente effettuare le operazioni elencate di seguito.

1. Prendere il nome della classe evento ed eliminare la parola "Event", aggiungendo poi la parola "Listener" a ciò che rimane. Questa è l'interfaccia listener che dovrete implementare nella vostra classe interna.
2. Implementare questa interfaccia e scrivere i metodi per gli eventi da intercettare. Per esempio, potreste voler gestire i movimenti del mouse, quindi



scrivrete il codice per il metodo **mouseMoved()** dell'interfaccia **MouseMotionListener**. Naturalmente dovrete implementare anche gli altri metodi, tuttavia per questa operazione spesso potrete usufruire di una scoria, come vedrete tra breve.

3. Creare un oggetto della classe listener implementata nel punto 2. Registratela con il vostro componente, ricorrendo al metodo prodotto premettendo “**add**” al nome del vostro listener, per esempio **addMouseMotionListener()**.

Di seguito sono elencate alcune interfacce listener.

<i>Interfaccia listener con adattatore</i>	<i>Metodi nell'interfaccia</i>
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)



Non è un elenco completo, anche perché il modello a eventi vi consente di creare dei tipi di evento personalizzati e i relativi listener. Troverete spesso librerie che hanno "inventato" i propri eventi, e la conoscenza acquisita in questo capitolo vi permetterà di comprendere come servirvene.

### Semplificazione mediante gli adattatori di listener

Nella tabella precedente avrete notato che alcune interfacce listener hanno un solo metodo. La loro implementazione è banale, tuttavia le interfacce listener dotate di metodi multipli possono essere meno semplici da utilizzare. Per esempio, se volete intercettare un clic del mouse (che non sia già gestito, per esempio, da un pulsante), dovrete scrivere un metodo per **mouseClicked()**. Ma poiché **MouseListener** è un'interfaccia dovrete implementare tutti gli altri metodi, anche se non eseguono alcunché. E questa è decisamente un'attività spiacente.

Per risolvere il problema, alcune (ma non tutte) interfacce listener che hanno più di un metodo sono fornite di *adattatori*, i nomi dei quali sono elencati nella tabella precedente. Ogni adattatore mette a disposizione metodi vuoti predefiniti corrispondenti a quelli previsti dall'interfaccia; quando create classi ereditate dall'adattatore, quindi, dovete sovrascrivere soltanto i metodi da modificare. Per esempio, una tipica classe **MouseListener** potrebbe essere la seguente:

```
class MyMouseListener extends MouseAdapter {  
    public void mouseClicked(MouseEvent e) {  
        // Reagisce al clic del mouse...  
    }  
}
```

L'obiettivo primario degli adattatori è semplificare la creazione delle classi listener.

Tuttavia gli adattatori presentano un inconveniente, abbastanza facile da incontrare. Supponete di scrivere un **MouseAdapter** come quello precedente:

```
class MyMouseListener extends MouseAdapter {  
    public void mouseClicked(MouseEvent e) {  
        // Reagisce al clic del mouse...  
    }  
}
```



Questo metodo non funziona, ma scoprirne i motivi non è così semplice: infatti, tutto compilerà e sarà eseguito alla perfezione, tranne che il metodo non verrà chiamato a seguito di un clic del mouse. Riuscite a individuare il problema? È nel nome del metodo, **MouseClicked()** invece di **mouseClicked()**. Un semplice cambiamento di maiuscola fa sì che il metodo venga aggiunto, perché considerato come nuovo. Tuttavia questo non è il metodo chiamato quando si fa clic con il mouse, pertanto non produce i risultati previsti. Malgrado questo inconveniente, un'interfaccia garantirà che i metodi saranno eseguiti correttamente.

Un metodo alternativo e migliore per garantire l'effettiva sovrascrittura di un metodo consiste nell'utilizzo dell'annotazione nativa **@Override** nel codice precedente.

**Esercizio 9 (5)** Cominciando da **ShowAddListeners.java** create un programma con la funzionalità completa di **typeinfo>ShowMethods.java**.

### ***Monitoraggio di eventi multipli***

Per dimostrare che questi eventi stanno effettivamente verificandosi, è opportuno creare un programma che tenga traccia del comportamento in un **JButton** quando viene premuto. Questo esempio vi mostra anche come ereditare il vostro oggetto-pulsante da **JButton**.<sup>6</sup>

Nel codice seguente la classe **MyButton** è una classe interna di **TrackEvent**, in modo che **MyButton** possa accedere alla finestra genitore e gestirne i campi di testo, requisito essenziale per registrare le informazioni sullo stato nei campi dell'oggetto genitore. Ovviamente questa è una soluzione limitata, poiché **MyButton** può essere utilizzato soltanto in abbinamento a **TrackEvent**. Questo genere di codice viene talvolta definito con l'espressione “a elevato abbinamento” (*highly coupled*):

```
//: gui/TrackEvent.java
// Mostra gli eventi quando si verificano.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;
```

6. In Java 1.0/1.1 non era possibile ereditare dall'oggetto-pulsante; questa è soltanto una delle numerose carenze progettuali che hanno caratterizzato queste versioni di Java.



```
public class TrackEvent extends JFrame {  
    private HashMap<String,JTextField> h =  
        new HashMap<String,JTextField>();  
    private String[] event = {  
        "focusGained", "focusLost", "keyPressed",  
        "keyReleased", "keyTyped", "mouseClicked",  
        "mouseEntered", "mouseExited", "mousePressed",  
        "mouseReleased", "mouseDragged", "mouseMoved"  
    };  
    private MyButton  
        b1 = new MyButton(Color.BLUE, "test1"),  
        b2 = new MyButton(Color.RED, "test2");  
    class MyButton extends JButton {  
        void report(String field, String msg) {  
            h.get(field).setText(msg);  
        }  
        FocusListener f1 = new FocusListener() {  
            public void focusGained(FocusEvent e) {  
                report("focusGained", e paramString());  
            }  
            public void focusLost(FocusEvent e) {  
                report("focusLost", e paramString());  
            }  
        };  
        KeyListener k1 = new KeyListener() {  
            public void keyPressed(KeyEvent e) {  
                report("keyPressed", e paramString());  
            }  
            public void keyReleased(KeyEvent e) {  
                report("keyReleased", e paramString());  
            }  
            public void keyTyped(KeyEvent e) {  
                report("keyTyped", e paramString());  
            }  
        };  
        MouseListener m1 = new MouseListener() {  
            public void mouseClicked(MouseEvent e) {  
                if (e getAction() == MouseEvent.BUTTON1)  
                    report("mouseClicked", e paramString());  
            }  
        };  
    }  
}
```



```
        report("mouseClicked", e paramString());
    }
    public void mouseEntered(MouseEvent e) {
        report("mouseEntered", e paramString());
    }
    public void mouseExited(MouseEvent e) {
        report("mouseExited", e paramString());
    }
    public void mousePressed(MouseEvent e) {
        report("mousePressed", e paramString());
    }
    public void mouseReleased(MouseEvent e) {
        report("mouseReleased", e paramString());
    }
}
MouseMotionListener mml = new MouseMotionListener() {
    public void mouseDragged(MouseEvent e) {
        report("mouseDragged", e paramString());
    }
    public void mouseMoved(MouseEvent e) {
        report("mouseMoved", e paramString());
    }
}
public MyButton(Color color, String label) {
    super(label);
    setBackground(color);
    addFocusListener(f1);
    addKeyListener(k1);
    addMouseListener(m1);
    addMouseMotionListener(mml);
}
}
public TrackEvent() {
    setLayout(new GridLayout(event.length + 1, 2));
    for(String evt : event) {
        JTextField t = new JTextField();
        t.setEditable(false);
```



```
        add(new JLabel(evt, JLabel.RIGHT));
        add(t);
        h.put(evt, t);
    }
    add(b1);
    add(b2);
}
public static void main(String[] args) {
    run(new TrackEvent(), 700, 500);
}
} //:-~
```

Nel costruttore **MyButton**, il colore del pulsante è impostato per mezzo di una chiamata a **SetBackground()**; tutti i listener sono installati mediante semplici chiamate di metodo.

La classe **TrackEvent** contiene un'**HashMap** per contenere le stringhe che rappresentano il tipo di evento, e i campi **JTextField** in cui sono contenute le informazioni su quell'evento. Naturalmente questi potrebbero essere creati in modo statico anziché inserendoli in un'**HashMap**, tuttavia l'autore ritiene che questa tecnica sia più facilmente utilizzabile e modificabile. In particolare, se dovete aggiungere o eliminare un nuovo tipo di evento in **TrackEvent**, vi basterà semplicemente integrare o rimuovere una stringa nell'array **event**, e tutto verrà eseguito in modo automatico.

Nella chiamata a **report()** vengono passati il nome dell'evento e la stringa di parametro dell'evento. Il metodo **report()** si serve dell'**HashMap** **h** nella classe esterna per ricercare il campo **JTextField** associato a quel nome di evento, e dispone la stringa di parametri in quel campo. Troverete interessante eseguire esperimenti con questo esempio, perché vi consente di vedere ciò che sta realmente accadendo con gli eventi del vostro programma.

**Esercizio 10** (6) Create un'applicazione utilizzando **SwingConsole**, con un **JButton** e un **JTextField**. Scrivete il listener adatto e connettetelo, in modo che quando il pulsante ha il fuoco i caratteri digitati compaiano in **JTextField**.

**Esercizio 11** (4) Create un nuovo tipo di pulsante ereditando da **JButton**. Ogni volta che premete questo pulsante esso dovrà cambiare colore, impostato a un valore casuale. Per un esempio di come sia possibile generare un valore casuale di colore, esaminate il codice di **ColorBoxes.java**, nel prosieguo del capitolo.



**Esercizio 12 (4)** Monitorate un nuovo tipo di evento in **TrackEvent**. java aggiungendo nuovo codice per la gestione degli eventi. Dovete determinare voi stessi il tipo di evento da controllare.

## Selezione di componenti Swing

Ora che avete compreso il funzionamento dei layout manager e del modello a eventi, siete pronti per iniziare a utilizzare i componenti Swing. Questa sezione presenta una panoramica, per quanto incompleta, dei componenti e delle caratteristiche di Swing che probabilmente avrete occasione di utilizzare con maggiore frequenza. Ogni esempio è stato concepito per essere ragionevolmente ridotto, in modo che possiate impiegarne il codice nei vostri programmi.

Tenete presente gli aspetti elencati di seguito.

1. Per osservare come si comporta ciascuno di questi esempi in fase di esecuzione, compilate ed eseguite il codice sorgente del capitolo, che potete scaricare da [www.mindview.net](http://www.mindview.net).
2. La documentazione JDK presente su <http://java.sun.com> esamina in modo completo tutte le classi e i metodi della libreria Swing: in questo capitolo ne viene trattato soltanto un sottoinsieme.
3. Come si è detto, grazie alla convenzione di nomenclatura utilizzata per gli eventi Swing, non è difficile intuire come scrivere e installare un gestore per un particolare tipo di evento. Servitevi del programma di consultazione **ShowAddListeners.java** presentato in questo capitolo come supporto alla ricerca di un determinato componente.
4. Quando le interfacce grafiche iniziano a essere complesse, dovreste considerare l'utilizzo di un ambiente di sviluppo visuale.

### Pulsanti

Swing mette a disposizione diversi tipi di pulsanti. Tutti i pulsanti, le caselle di scelta, i pulsanti radio e persino le voci di menu sono ereditati da **AbstractButton**. Esaminerete tra breve le voci di menu; l'esempio seguente mostra i vari tipi di pulsanti disponibili:

```
//: gui/Buttons.java
// Vari tipi di pulsanti Swing.
import javax.swing.*;
import javax.swing.border.*;
```



```
import javax.swing.plaf.basic.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Buttons extends JFrame {
    private JButton jb = new JButton("JButton");
    private BasicArrowButton
        up = new BasicArrowButton(BasicArrowButton.NORTH),
        down = new BasicArrowButton(BasicArrowButton.SOUTH),
        right = new BasicArrowButton(BasicArrowButton.EAST),
        left = new BasicArrowButton(BasicArrowButton.WEST);
    public Buttons() {
        setLayout(new FlowLayout());
        add(jb);
        add(new JToggleButton("JToggleButton"));
        add(new JCheckBox("JCheckBox"));
        add(new JRadioButton("JRadioButton"));
        JPanel jp = new JPanel();
        jp.setBorder(new TitledBorder("Directions"));
        jp.add(up);
        jp.add(down);
        jp.add(left);
        jp.add(right);
        add(jp);
    }
    public static void main(String[] args) {
        run(new Buttons(), 350, 200);
    }
} //:~
```

Il programma inizia con l'inserimento di **BasicArrowButton** da **javax.swing.plaf.basic**, poi continua con i vari tipi di pulsanti specifici. Eseguendo questo codice vedrete che il pulsante **JToggleButton** mantiene l'ultima posizione, che sia premuto o meno. Le caselle di scelta e i pulsanti radio si comportano in modo identico, assumendo lo stato attivo o disattivo: entrambi sono ereditati da **JToggleButton**.



## Gruppi di pulsanti

Se desiderate pulsanti radio con un comportamento “autoescludente” (*or esclusivo*) dovrete aggiungerli a un gruppo di pulsanti (“button group”). Tuttavia qualsiasi **AbstractButton** può essere aggiunto a un oggetto **ButtonGroup**, come dimostra l’esempio seguente.

Allo scopo di evitare la ripetizione del codice, per generare gruppi con diversi tipi di pulsanti viene utilizzata la tecnica di riflessione. Questa soluzione è offerta dal metodo **makeBPanel()**, che crea un gruppo di pulsanti in un **JPanel**; il secondo argomento da fornire a **makeBPanel()** è un array di **String**. Per ogni **String**, a **JPanel** viene aggiunto un pulsante della classe indicata dal primo argomento:

```
//: gui/ButtonGroups.java
// Utilizza la riflessione per creare gruppi
// con diversi tipi di AbstractButton.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingConsole.*;

public class ButtonGroups extends JFrame {
    private static String[] ids = {
        "June", "Ward", "Beaver", "Wally", "Eddie", "Lumpy"
    };
    static JPanel makeBPanel(
        Class<? extends AbstractButton> kind, String[] ids) {
        ButtonGroup bg = new ButtonGroup();
        JPanel jp = new JPanel();
        String title = kind.getName();
        title = title.substring(title.lastIndexOf('.') + 1);
        jp.setBorder(new TitledBorder(title));
        for(String id : ids) {
            AbstractButton ab = new JButton("Failed");
            try {
                // Ottiene il costruttore dinamico che accetta un
                // argomento String:
```



```
Constructor ctor =
    kind.getConstructor(String.class);
    // Crea un nuovo oggetto:
    ab = (AbstractButton)ctor.newInstance(id);
} catch(Exception ex) {
    System.err.println("can't create " + kind);
}
bg.add(ab);
jp.add(ab) ;
}
return jp;
}
public ButtonGroups() {
    setLayout(new FlowLayout());
    add(makeBPanel(JButton.class, ids));
    add(makeBPanel(JToggleButton.class, ids));
    add(makeBPanel(JCheckBox.class, ids));
    add(makeBPanel(JRadioButton.class, ids));
}
public static void main(String[] args) {
    run(new ButtonGroups(), 600, 350);
}
} //:~
```

Il titolo della finestra corrisponde al nome della classe, da cui sono eliminate tutte le informazioni relative al percorso. **AbstractButton** è inizializzato a un **JButton** con l'etichetta “Failed”: in tal modo, anche se ignorate il messaggio di un'eccezione il problema verrà comunque visualizzato.

Il metodo **getConstructor()** restituisce un oggetto **Constructor**, che accetta l'array di argomenti dei tipi nella lista di **Class** passata a **getConstructor()**. A questo punto non dovete fare altro che chiamare il metodo **newInstance()** passandogli un elenco di argomenti, in questo caso le **String** presenti nell'array **ids**. Per realizzare il meccanismo di “autoesclusione” create un gruppo di pulsanti e aggiungetevi ogni pulsante cui desiderate assegnare questo comportamento; eseguendo il programma, vedrete che tale funzionalità sarà implementata in tutti i pulsanti tranne **JButton**.



## Icone

Potete utilizzare un oggetto **Icon** all'interno di un'etichetta **JLabel** o di qualsiasi oggetto erediti da **AbstractButton**, compresi **JButton**, **JCheckBox**, **JRadioButton** e i diversi tipi di **JMenuItem**. L'utilizzo di oggetti **Icon** con **JLabel** è abbastanza immediato, come vedrete in un esempio successivo. L'esempio seguente illustra tutte le modalità di impiego degli oggetti **Icon** con i pulsanti e i loro discendenti.

Potete utilizzare qualsiasi immagine **GIF**; quelle adottate nell'esempio fanno parte del codice a corredo di questo manuale, disponibile su [www.mindview.net](http://www.mindview.net). Per aprire un file e caricare l'immagine è sufficiente che creiate un'**ImageIcon** e forniate il nome del file: potrete poi utilizzare l'oggetto **Icon** nel vostro programma.

```
//: gui/Faces.java
// Comportamento di Icon con JButton.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Faces extends JFrame {
    private static Icon[] faces;
    private JButton jb, jb2 = new JButton("Disable");
    private boolean mad = false;
    public Faces() {
        faces = new Icon[]{
            new ImageIcon(getClass().getResource("Face0.gif")),
            new ImageIcon(getClass().getResource("Face1.gif")),
            new ImageIcon(getClass().getResource("Face2.gif")),
            new ImageIcon(getClass().getResource("Face3.gif")),
            new ImageIcon(getClass().getResource("Face4.gif"))
        };
        jb = new JButton("JButton", faces[3]);
        setLayout(new FlowLayout());
        jb.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if(mad) {
```



```
        jb.setIcon(faces[3]);
        mad = false;
    } else {
        jb.setIcon(faces[0]);
        mad = true;
    }
    jb.setVerticalAlignment(JButton.TOP);
    jb.setHorizontalAlignment(JButton.LEFT);
}
});
jb.setRolloverEnabled(true);
jb.setRolloverIcon(faces[1]);
jb.setPressedIcon(faces[2]);
jb.setDisabledIcon(faces[4]);
jb.setToolTipText("Wow!");
add(jb);
jb2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(jb.isEnabled()) {
            jb.setEnabled(false);
            jb2.setText("Enable");
        } else {
            jb.setEnabled(true);
            jb2.setText("Disable");
        }
    }
});
add(jb2);
}
public static void main(String[] args) {
    run(new Faces(), 250, 125);
}
}
//:~
```

Un'Icon può essere utilizzata come argomento per diversi costruttori di componenti Swing, ed è anche possibile aggiungerla o modificarla per mezzo del metodo **setIcon()**. L'esempio precedente mostra anche che un JButton, come



qualsiasi **AbstractButton**, può impostare diverse icone che verranno visualizzate al verificarsi di determinati eventi: pulsante premuto, disabilitato, o effetto “roll over”, vale a dire quando il cursore del mouse si sposta sul controllo, senza che l’utente faccia clic. Come potete vedere questa funzionalità dà al pulsante un gradevole aspetto animato.

### **Tooltip**

Nell’esempio precedente è stato aggiunto un suggerimento (“tooltip”) al pulsante. Quasi tutte le classi che utilizzerete per creare le interfacce grafiche sono derivate da **JComponent**, che offre un metodo chiamato **setToolTipText(String)**. Pertanto potete impostare un tooltip praticamente per qualunque oggetto posizionate sul vostro form, come dimostra questo esempio, che si riferisce a un ipotetico oggetto **jc** di qualsiasi classe derivata da **JComponent**:

```
jc.setToolTipText("My tip");
```

In questo modo, quando il mouse si posiziona sopra il componente **JComponent** per un breve periodo di tempo, accanto al puntatore del mouse compare una piccola casella che visualizza il suggerimento.

### **Campi di testo**

Questo esempio illustra le potenzialità del componente **JTextField**:

```
//: gui/TextFields.java
// TextField ed eventi Java.
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;
```

```
public class TextFields extends JFrame {
    private JButton
        b1 = new JButton("Get Text"),
        b2 = new JButton("Set Text");
}
```



```
private JTextField  
    t1 = new JTextField(30),  
    t2 = new JTextField(30),  
    t3 = new JTextField(30);  
  
private String s = "";  
private UpperCaseDocument ucd = new UpperCaseDocument();  
public TextFields() {  
    t1.setDocument(ucd);  
    ucd.addDocumentListener(new T1());  
    b1.addActionListener(new B1());  
    b2.addActionListener(new B2());  
    t1.addActionListener(new T1A());  
    setLayout(new FlowLayout());  
    add(b1);  
    add(b2);  
    add(t1);  
    add(t2);  
    add(t3);  
}  
class T1 implements DocumentListener {  
    public void changedUpdate(DocumentEvent e) {}  
    public void insertUpdate(DocumentEvent e) {  
        t2.setText(t1.getText());  
        t3.setText("Text " + t1.getText());  
    }  
    public void removeUpdate(DocumentEvent e) {  
        t2.setText(t1.getText());  
    }  
}  
class T1A implements ActionListener {  
    private int count = 0;  
    public void actionPerformed(ActionEvent e) {  
        t3.setText("t1 Action Event " + count++);  
    }  
}  
class B1 implements ActionListener {
```



```
public void actionPerformed(ActionEvent e) {
    if(tl.getSelectedText() == null)
        s = tl.getText();
    else
        s = tl.getSelectedText();
    tl.setEditable(true);
}
}

class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        ucd.setUpperCase(false);
        tl.setText("Inserted by Button 2: " + s);
        ucd.setUpperCase(true);
        tl.setEditable(false);
    }
}

public static void main(String[] args) {
    run(new TextFields(), 375, 200);
}
}

class UpperCaseDocument extends PlainDocument {
    private boolean upperCase = true;
    public void setUpperCase(boolean flag) {
        upperCase = flag;
    }
    public void
    insertString(int offset, String str, AttributeSet attSet)
    throws BadLocationException {
        if(upperCase) str = str.toUpperCase();
        super.insertString(offset, str, attSet);
    }
} ///:~
```

L'oggetto **JTextField t3** è incluso come punto di riferimento per segnalare l'attivazione dell'action listener del **JtextField t1**. Come vedrete, l'ActionListener per un **JTextField** viene attivato soltanto premendo il tasto Invio.



Il componente **JTextField t1** è collegato ad alcuni listener. Il listener **T1** è un **DocumentListener** che risponde a qualsiasi modifica del “documento”, in questo caso sul contenuto del campo **JTextField**: **T1** copia automaticamente tutto il testo da **t1** in **t2**. Inoltre, il documento contenuto in **t1** viene impostato a una classe derivata da **PlainDocument**, chiamata **UpperCaseDocument**, che trasforma tutti i caratteri in maiuscolo. Questa classe rileva automaticamente la pressione del tasto Backspace, spostando il cursore del testo ed eseguendo la cancellazione nel modo convenzionale.

**Esercizio 13 (3)** Modificate **TextFields.java** in modo che i caratteri in **t2** non vengano convertiti automaticamente in maiuscolo.

### *Bordi*

**JComponent** contiene un metodo chiamato **setBorder()**, che vi permette di disporre vari tipi di bordo su qualsiasi componente visibile. L'esempio seguente mostra il funzionamento di alcuni dei bordi disponibili, utilizzando il metodo **showBorder()** che crea un oggetto **JPanel** e vi inserisce un bordo. In questo esempio ci si serve anche delle funzionalità RTTI per identificare il nome del bordo in utilizzo, omettendo le informazioni su percorso. Il nome del bordo viene poi inserito in una **JLabel** posta al centro del pannello:

```
//: gui/Borders.java
// Diversi bordi Swing.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Borders extends JFrame {
    static JPanel showBorder(Border b) {
        JPanel jp = new JPanel();
        jp.setLayout(new BorderLayout());
        String nm = b.getClass().toString();
        nm = nm.substring(nm.lastIndexOf('.') + 1);
        jp.add(new JLabel(nm, JLabel.CENTER),
               BorderLayout.CENTER);
        jp.setBorder(b);
        return jp;
    }
}
```



```
public Borders() {  
    setLayout(new GridLayout(2,4));  
    add(showBorder(new TitledBorder("Title")));  
    add(showBorder(new EtchedBorder()));  
    add(showBorder(new LineBorder(Color.BLUE)));  
    add(showBorder(  
        new MatteBorder(5,5,30,30,Color.GREEN)));  
    add(showBorder(  
        new BevelBorder(BevelBorder.RAISED)));  
    add(showBorder(  
        new SoftBevelBorder(BevelBorder.LOWERED)));  
    add(showBorder(new CompoundBorder(  
        new EtchedBorder(),  
        new LineBorder(Color.RED))));  
}  
public static void main(String[] args) {  
    run(new Borders(), 500, 300);  
}  
} ///:-
```

Potete anche creare bordi personalizzati e inserirli in pulsanti, etichette ecc.: in pratica, in qualsiasi componente derivato da **JComponent**.

### ***Un mini-editor***

Il controllo **JTextPane** fornisce un ottimo supporto per la modifica del testo senza richiedere molto impegno al programmatore. L'esempio seguente fa un utilizzo molto semplice di questo componente, trascurando la maggior parte delle sue funzionalità:

```
//: gui/TextPane.java  
// Il controllo JTextPane e' un piccolo editor di testi.  
import javax.swing.*;  
import java.awt.*;  
import java.awt.event.*;  
import net.mindview.util.*;  
import static net.mindview.util.SwingConsole.*;
```



```
public class TextPane extends JFrame {  
    private JButton b = new JButton("Add Text");  
    private JTextPane tp = new JTextPane();  
    private static Generator sg =  
        new RandomGenerator.String(7);  
    public TextPane() {  
        b.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                for(int i = 1; i < 10; i++)  
                    tp.setText(tp.getText() + sg.next() + "\n");  
            }  
        });  
        add(new JScrollPane(tp));  
        add(BorderLayout.SOUTH, b);  
    }  
    public static void main(String[] args) {  
        run(new TextPane(), 475, 425);  
    }  
}
```

Il pulsante aggiunge testo generato casualmente. Considerate tuttavia che lo scopo del componente **JTextPane** è consentire la modifica del testo, di conseguenza non è presente alcun metodo **append()**.

In questo esempio, che come si è detto rappresenta un utilizzo molto limitato delle potenzialità del componente **JTextPane**, il testo deve essere selezionato, modificato e poi nuovamente disposto nel pannello servendosi del metodo **setText()**.

Gli elementi vengono aggiunti a **JFrame** utilizzando il layout predefinito **BorderLayout**. Il **JTextPane**, posto all'interno di un **JScrollPane**, è aggiunto al **JFrame** senza indicare un'area specifica, e per questo motivo viene inserito nell'area centrale. Il pulsante **JButton** è invece aggiunto all'area **SOUTH**.

Notate le caratteristiche native di **JTextPane**, in particolare l'“a capo automatico”. Nella documentazione JDK troverete l'elenco completo delle funzionalità di questo componente.

**Esercizio 14** (2) Modificate l'esempio **TextPane.java** per utilizzare un componente **JTextArea** in luogo di **JTextPane**.



## Caselle di scelta

Una casella di scelta, chiamata anche casella di controllo, è un meccanismo per eseguire singole scelte di tipo vero/falso, composto di un piccolo riquadro e di un'etichetta: la casella può essere vuota (deselezionata) oppure contenere una piccola "x" o un segno di spunta (selezionata).

Di norma la creazione di una **JCheckBox** avviene mediante un costruttore che accetta come argomento l'etichetta da visualizzare. Se desiderate intervenire sull'etichetta dopo la creazione della **JCheckBox**, potete ottenere e impostare lo stato del controllo, nonché il valore dell'etichetta.

Ogni volta che una **JCheckBox** viene selezionata o deselectata si verifica un evento che può essere gestito in modo analogo a quanto avviene con un pulsante, cioè tramite un **ActionListener**. L'esempio seguente si serve di un componente **JTextArea** per elencare tutte le caselle di scelta selezionate:

```
//: gui/CheckBoxes.java
// Uso del componente JCheckBox.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class CheckBoxes extends JFrame {
    private JTextArea t = new JTextArea(6, 15);
    private JCheckBox
        cb1 = new JCheckBox("Check Box 1"),
        cb2 = new JCheckBox("Check Box 2"),
        cb3 = new JCheckBox("Check Box 3");
    public CheckBoxes() {
        cb1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                trace("1", cb1);
            }
        });
        cb2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                trace("2", cb2);
            }
        });
    }
}
```



```
});  
cb3.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        trace("3", cb3);  
    }  
});  
setLayout(new FlowLayout());  
add(new JScrollPane(t));  
add(cb1);  
add(cb2);  
add(cb3);  
}  
private void trace(String b, JCheckBox cb) {  
    if(cb.isSelected())  
        t.append("Box " + b + " Set\n");  
    else  
        t.append("Box " + b + " Cleared\n");  
}  
public static void main(String[] args) {  
    run(new CheckBoxes(), 200, 300);  
}  
} //:~
```

Il metodo **trace()** invia il nome della **JCheckBox** selezionata e il relativo stato al componente **JTextArea** utilizzando il metodo **append()**; la casella di testo, quindi, visualizzerà un elenco dei nomi delle caselle di scelta che sono state selezionate, associate al loro stato.

**Esercizio 15 (5)** Aggiungete una casella di scelta all'applicazione creata nell'esercizio 5, poi intercettate l'evento e inserite un testo diverso nella casella di testo.

### **Pulsanti radio**

L'idea dei pulsanti radio impiegati nella programmazione delle interfacce grafiche ha origine dalle vecchie radio con tasti meccanici: premendone uno, tutti gli altri tasti si disattivavano, scattando verso l'alto. Questo tipo di componente "autoescludente", quindi, permette di eseguire una sola selezione tra più opzioni.



Per impostare un gruppo di **JRadioButton** associati, dovete aggiungere questi componenti a un **ButtonGroup**: ricordate che in ogni form può essere presente qualsiasi numero di **ButtonGroup**. Uno solo dei pulsanti appartenenti a un gruppo può essere impostato al valore **true**, mediante il secondo argomento del costruttore; se tentate di impostare più di un pulsante di scelta a **true**, soltanto l'ultimo assumerà questo valore.

L'esempio seguente illustra l'impiego dei pulsanti radio e l'intercettazione degli eventi per mezzo di un **ActionListener**:

```
//: gui/RadioButtons.java
// Utilizzo del componente JRadioButton.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class RadioButtons extends JFrame {
    private JTextField t = new JTextField(15);
    private ButtonGroup g = new ButtonGroup();
    private JRadioButton
        rb1 = new JRadioButton("one", false),
        rb2 = new JRadioButton("two", false),
        rb3 = new JRadioButton("three", false);
    private ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText("Radio button " +
                ((JRadioButton)e.getSource()).getText());
        }
    };
    public RadioButtons() {
        rb1.addActionListener(al);
        rb2.addActionListener(al);
        rb3.addActionListener(al);
        g.add(rb1); g.add(rb2); g.add(rb3);
        t.setEditable(false);
        setLayout(new FlowLayout());
        add(t);
    }
}
```



```
        add(rb1);
        add(rb2);
        add(rb3);
    }
    public static void main(String[] args) {
        run(new RadioButtons(), 200, 125);
    }
} //:~
```

Per visualizzare lo stato dei controlli è utilizzata una casella di testo, impostata a "non modificabile" in quanto deve limitarsi a visualizzare i dati. Questa tecnica è un'alternativa all'impiego del controllo **JLabel**.

### Caselle combinate o elenchi a discesa

Come un gruppo di pulsanti radio, una casella combinata rappresenta un metodo per costringere l'utente a selezionare un solo elemento da un gruppo di opzioni. Tuttavia una casella combinata è più compatta di un gruppo di pulsanti, ed è più facile modificare gli elementi che compongono l'elenco, senza sorprendere l'utente. È certamente possibile modificare i pulsanti radio, tuttavia il risultato sarebbe poco estetico.

In modalità predefinita il comportamento della casella combinata **JComboBox** non è come quello dell'omonimo controllo presente negli ambienti grafici della maggior parte dei sistemi operativi, che permette *sia* di scegliere da un elenco *sia* di digitare la selezione: per emulare questo comportamento dovete chiamare il metodo **setEditable()**. Con una casella **JComboBox** è possibile scegliere un solo elemento. Nell'esempio seguente la casella di **JComboBox** si attiva con un certo numero di voci, che vengono poi integrate da altri elementi quando si preme il pulsante.

```
//: gui/ComboBoxes.java
// Utilizzo delle caselle a discesa.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;
```

```
public class ComboBoxes extends JFrame {
    private String[] description = {
```



```
"Ebullient", "Obtuse", "Recalcitrant", "Brilliant",
"Somnescient", "Timorous", "Florid", "Putrescent"
};

private JTextField t = new JTextField(15);
private JComboBox c = new JComboBox();
private JButton b = new JButton("Add items");
private int count = 0;
public ComboBoxes() {
    for(int i = 0; i < 4; i++)
        c.addItem(description[count++]);
    t.setEditable(false);
    b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            if(count < description.length)
                c.addItem(description[count++]);
        }
    });
    c.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText("index: " + c.getSelectedIndex() + " " +
                ((JComboBox)e.getSource()).getSelectedItem());
        }
    });
    setLayout(new FlowLayout());
    add(t);
    add(c);
    add(b);
}
public static void main(String[] args) {
    run(new ComboBoxes(), 200, 175);
}
} ///:-
```

La casella **JTextField** visualizza l'indice, vale a dire il numero progressivo dell'elemento selezionato in quel momento nella casella combinata, e il testo associato.



## Caselle di riepilogo

Le caselle di riepilogo non sono diverse dalle caselle **JComboBox** soltanto nell'aspetto esteriore. Mentre una casella combinata **JComboBox** "si apre" quando viene attivata, una **JList** occupa sempre un certo numero di righe e non cambia dimensione. Per visualizzare gli elementi selezionati nella casella di riepilogo, dovete semplicemente chiamare **getSelectedValues()**, che restituisce un array di **String** contenente gli elementi selezionati.

**JList** permette di eseguire selezioni multiple mediante la combinazione Ctrl+clic, che si ottiene tenendo premuto il tasto Ctrl e contemporaneamente facendo clic con il mouse su una voce dell'elenco: l'elemento selezionato rimarrà tale e potrete selezionare quanti elementi vi occorrono. Se selezionate un elemento ed eseguite, in modo analogo a quanto già descritto, la combinazione Maiusc+clic su un altro elemento, verranno selezionati tutti gli elementi presenti nell'intervallo tra le due voci; per rimuovere un elemento da una selezione multipla, potete utilizzare la combinazione Ctrl+clic.

```
//: gui/List.java
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class List extends JFrame {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    private DefaultListModel lItems = new DefaultListModel();
    private JList lst = new JList(lItems);
    private JTextArea t =
        new JTextArea(flavors.length, 20);
    private JButton b = new JButton("Add Item");
    private ActionListener bl = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
```



```
        if(count < flavors.length) {
            lItems.addElement(flavors[count++]);
        } else {
            // Disabilita, perche' non vi sono altri sapori da
            // aggiungere all'elenco
            b.setEnabled(false);
        }
    };
private ListSelectionListener ll =
new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
        if(e.getValueIsAdjusting()) return;
        t.setText("");
        for(Object item : lst.getSelectedValues())
            t.append(item + "\n");
    }
};
private int count = 0;
public List() {
    t.setEditable(false);
    setLayout(new FlowLayout());
    // Crea i bordi per i componenti:
    Border brd = BorderFactory.createMatteBorder(
        1, 1, 2, 2, Color.BLACK);
    lst.setBorder(brd);
    t.setBorder(brd);
    // Aggiunge i primi quattro sapori alla List
    for(int i = 0; i < 4; i++)
        lItems.addElement(flavors[count++]);
    add(t);
    add(lst);
    add(b);
    // Registra i listener degli eventi
    lst.addListSelectionListener(ll);
    b.addActionListener(bl);
}
```



```
public static void main(String[] args) {
    run(new List(), 250, 375);
}
} //:~
```

Come vedete, alla **JList** e alla casella di testo sono stati aggiunti bordi.

Per inserire un array di **String** in una **JList** esiste una soluzione molto più semplice: è sufficiente passare l'array al costruttore di **JList** che popolerà l'elenco automaticamente. In questo esempio, l'unico motivo per utilizzare un oggetto della classe **DefaultListModel** è la possibilità di manipolare la **JList** in fase di esecuzione del programma.

Il componente **JList** non fornisce automaticamente il supporto allo scorrimento; se lo ritenete necessario, basterà che includiate **JList** in un oggetto **JScrollPane**: i dettagli verranno gestiti automaticamente dal programma.

**Esercizio 16** (5) Semplificate **List.java** passando l'array al costruttore ed eliminando l'aggiunta dinamica degli elementi.

### Pannelli a schede

Il componente **JTabbedPane** vi consente di realizzare un “pannello a schede” nel quale le singole schede sono allineate lungo i bordi del controllo e identificate da una linguetta, come avviene in uno schedario o in una rubrica cartacea: selezionando la linguetta (*tab*) corrispondente alla scheda, il pannello porterà in primo piano la scheda relativa.

```
//: gui/TabbedPane1.java
// Dimostrazione del pannello a schede.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class TabbedPane1 extends JFrame {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
}
```



```
private JTabbedPane tabs = new JTabbedPane();
private JTextField txt = new JTextField(20);
public TabbedPane1() {
    int i = 0;
    for(String flavor : flavors)
        tabs.addTab(flavors[i],
            new JButton("Tabbed pane " + i++));
    tabs.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
            txt.setText("Tab selected: " +
            tabs.getSelectedIndex());
        }
    });
    add(BorderLayout.SOUTH, txt);
    add(tabs);
}
public static void main(String[] args) {
    run(new TabbedPane1(), 400, 250);
}
} //:-~
```

Eseguendo il programma vedrete che, qualora il numero di schede ecceda le dimensioni della finestra, il **JTabbedPane** le disporrà automaticamente su più file o in altro modo, a seconda del sistema operativo utilizzato. Potete osservare questo comportamento ridimensionando la finestra al momento di eseguire il programma.

### ***Finestre di messaggio***

Gli ambienti a finestre contengono generalmente un insieme standard di finestre di messaggio che permettono di visualizzare o di ottenere informazioni dall'utente, in modo molto rapido.

In Swing queste finestre di messaggio sono contenute nei controlli **JOptionPane**. Esistono diversi tipi di finestre di questo tipo, alcune molto elaborate; tuttavia quelle di utilizzo più comune sono le finestre informative e le finestre che richiedono conferma all'utente, richiamabili rispettivamente con i metodi **static JOptionPane.showMessageDialog()** e **JOptionPane.showConfirmDialog()**.



L'esempio seguente mostra un sottoinsieme delle finestre di messaggio disponibili con **JOptionPane**:

```
//: gui/MessageBoxes.java
// Dimostrazione del componente JOptionPane.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class MessageBoxes extends JFrame {
    private JButton[] b = {
        new JButton("Alert"), new JButton("Yes/No"),
        new JButton("Color"), new JButton("Input"),
        new JButton("3 Vals")
    };
    private JTextField txt = new JTextField(15);
    private ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String id = ((JButton)e.getSource()).getText();
            if(id.equals("Alert"))
                JOptionPane.showMessageDialog(null,
                    "There's a bug on you!", "Hey!",
                    JOptionPane.ERROR_MESSAGE);
            else if(id.equals("Yes/No"))
                JOptionPane.showConfirmDialog(null,
                    "or no", "choose yes",
                    JOptionPane.YES_NO_OPTION);
            else if(id.equals("Color")) {
                Object[] options = { "Red", "Green" };
                int sel = JOptionPane.showOptionDialog(
                    null, "Choose a Color!", "Warning",
                    JOptionPane.DEFAULT_OPTION,
                    JOptionPane.WARNING_MESSAGE, null,
                    options, options[0]);
                if(sel != JOptionPane.CLOSED_OPTION)
                    txt.setText("Color Selected: " + options[sel]);
            }
        }
    };
}
```



```
    } else if(id.equals("Input")) {
        String val = JOptionPane.showInputDialog(
            "How many fingers do you see?");
        txt.setText(val);
    } else if(id.equals("3 Vals")) {
        Object[] selections = {"Third", "Second", "First"};
        Object val = JOptionPane.showInputDialog(
            null, "Choose one", "Input",
            JOptionPane.INFORMATION_MESSAGE,
            null, selections, selections[0]);
        if(val != null)
            txt.setText(val.toString());
    }
}
public MessageBoxes() {
    setLayout(new FlowLayout());
    for(int i = 0; i < b.length; i++) {
        b[i].addActionListener(al);
        add(b[i]);
    }
    add(txt);
}
public static void main(String[] args) {
    run(new MessageBoxes(), 200, 200);
}
} //:-
```

Per scrivere un solo **ActionListener** l'autore ha utilizzato la tecnica, piuttosto rischiosa, che consiste nel controllare il valore delle etichette **String** sui pulsanti. Il problema di questo approccio è che nell'ottenimento dell'etichetta potrebbero verificarsi alcuni errori, in particolare per quanto riguarda le maiuscole e le minuscole, e tali errori sono molto difficili da individuare.

Notate che i metodi **showOptionDialog()** e **showInputDialog()** restituiscono gli oggetti contenenti il valore inserito dall'utente.

**Esercizio 17(5)** Create un'applicazione utilizzando **SwingConsole**. Nella documentazione di JDK (<http://java.sun.com>) cercate la classe **JPassword-**



**Field** e aggiungerela al programma. Se l'utente digita la parola d'accesso corretta, visualizzate un messaggio opportuno con **JOptionPane**.

**Esercizio 18 (4)** Modificate **MessageBoxes.java** in modo da avere un **ActionListener** specifico per ogni pulsante.

## Menu

Tutti i componenti in grado di contenere un menu, compresi **JApplet**, **JFrame**, **JDialog** e i loro discendenti, dispongono di un metodo **setJMenuBar()** che accetta una **JMenuBar**, una soltanto per ogni componente. Potete aggiungere **JMenu** alle **JMenuBar** e **JMenuItem** per le voci di menu abbinate ai **JMenu**; a ogni **JMenuItem** deve essere associato un **ActionListener**, che si attiverà alla selezione della voce di menu corrispondente. Con Java e Swing è necessario gestire l'assemblaggio dei menu direttamente nel codice sorgente. Considerate questo esempio molto semplice:

```
//: gui/SimpleMenus.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class SimpleMenus extends JFrame {
    private JTextField t = new JTextField(15);
    private ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText(((JMenuItem)e.getSource()).getText());
        }
    };
    private JMenu[] menus = {
        new JMenu("Winken"), new JMenu("Blinken"),
        new JMenu("Nod")
    };
    private JMenuItem[] items = {
        new JMenuItem("Fee"), new JMenuItem("Fi"),
        new JMenuItem("Fo"), new JMenuItem("Zip"),
        new JMenuItem("Zap"), new JMenuItem("Zot"),
        new JMenuItem("Olly"), new JMenuItem("Oxen"),
    };
}
```



```

        new JMenuItem("Free")
    };
public SimpleMenus() {
    for(int i = 0; i < items.length; i++) {
        items[i].addActionListener(al);
        menus[i % 3].add(items[i]);
    }
    JMenuBar mb = new JMenuBar();
    for(JMenu jm : menus)
        mb.add(jm);
    setJMenuBar(mb);
    setLayout(new FlowLayout());
    add(t);
}
public static void main(String[] args) {
    run(new SimpleMenus(), 200, 150);
}
} //:-

```

L'utilizzo dell'operatore modulo in “*i % 3*” distribuisce le voci di menu tra i tre **JMenu**. Ogni **JMenuItem** deve essere collegato a un **ActionListener**; in questo caso viene utilizzato un solo **ActionListener**, ma di norma occorre specificarne uno per ogni **JMenuItem**.

**JMenuItem** estende **AbstractButton**, pertanto possiede alcuni comportamenti tipici dei pulsanti. In sé, fornisce un elemento che può essere inserito in un menu a discesa. Esistono altri tre tipi che estendono **JMenuItem**: **JMenu**, utilizzato per contenere altri **JMenuItem**, che consente di creare menu a discesa, **JCheckBoxMenuItem**, che visualizza un segno di spunta per indicare se la voce corrispondente è selezionata, e **JRadioButtonMenuItem**, che contiene un pulsante radio.

L'esempio seguente è più specifico: per creare i menu sono stati utilizzati ancora una volta i gusti di gelato. L'esempio mostra il funzionamento dei menu a discesa, delle abbreviazioni da tastiera e degli oggetti **JCheckBoxMenuItem**, nonché una tecnica per modificare i menu in modo dinamico:

```

//: gui/Menu.java
// Sottomenu, voci di menu con check box, scambio di menu,
// abbreviazioni da tastiera e action command.

```



```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Menus extends JFrame {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Min Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    private JTextField t = new JTextField("No flavor", 30);
    private JMenuBar mb1 = new JMenuBar();
    private JMenu
        f = new JMenu("File"),
        m = new JMenu("Flavors"),
        s = new JMenu("Safety");
    // Approccio alternativo:
    private JCheckBoxMenuItem[] safety = {
        new JCheckBoxMenuItem("Guard"),
        new JCheckBoxMenuItem("Hide")
    };
    private JMenuItem[] file = { new JMenuItem("Open") };
    // Una seconda barra di menu da scambiare con la prima:
    private JMenuBar mb2 = new JMenuBar();
    private JMenu fooBar = new JMenu("fooBar");
    private JMenuItem[] other = {
        // L'aggiunta di abbreviazioni da tastiera è molto
        // semplice, ma queste possono essere passate
        // solo al costruttore di JMenuItem:
        new JMenuItem("Foo", KeyEvent.VK_F),
        new JMenuItem("Bar", KeyEvent.VK_A),
        // Nessuna abbreviazione da tastiera:
        new JMenuItem("Baz"),
    };
    private JButton b = new JButton("Swap Menus");
}
```



```
class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuBar m = getJMenuBar();
        setJMenuBar(m == mb1 ? mb2 : mb1);
        validate(); // Aggiorna il frame
    }
}
class ML implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuItem target = (JMenuItem)e.getSource();
        String actionCommand = target.getActionCommand();
        if(actionCommand.equals("Open")) {
            String s = t.getText();
            boolean chosen = false;
            for(String flavor : flavors)
                if(s.equals(flavor))
                    chosen = true;
            if(!chosen)
                t.setText("Choose a flavor first!");
            else
                t.setText("Opening " + s + ". Mmm, mmm!");
        }
    }
}
class FL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuItem target = (JMenuItem)e.getSource();
        t.setText(target.getText());
    }
}
// In alternativa, potete creare una classe diversa per
// ogni MenuItem, nel qual caso non avrete bisogno
// di determinare quale sia:
class Fool implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Foo selected");
    }
}
```



```
        }
    }
    class BarL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            t.setText("Bar selected");
        }
    }
    class BazL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            t.setText("Baz selected");
        }
    }
    class CMIL implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            JCheckBoxMenuItem target =
                (JCheckBoxMenuItem)e.getSource();
            String actionCommand = target.getActionCommand();
            if(actionCommand.equals("Guard"))
                t.setText("Guard the Ice Cream! " +
                    "Guarding is " + target.getState());
            else if(actionCommand.equals("Hide"))
                t.setText("Hide the Ice Cream! " +
                    "Is it hidden? " + target.getState());
        }
    }
    public Menus() {
        ML ml = new ML();
        CMIL cmil = new CMIL();
        safety[0].setActionCommand("Guard");
        safety[0].setMnemonic(KeyEvent.VK_G);
        safety[0].addItemListener(cmil);
        safety[1].setActionCommand("Hide");
        safety[1].setMnemonic(KeyEvent.VK_H);
        safety[1].addItemListener(cmil);
        other[0].addActionListener(new FooL());
        other[1].addActionListener(new BarL());
        other[2].addActionListener(new BazL());
```



```
FL f1 = new FL();
int n = 0;
for(String flavor : flavors) {
    JMenuItem mi = new JMenuItem(flavor);
    mi.addActionListener(f1);
    m.add(mi);
    // Aggiunge dei separatori a intervalli:
    if((n++ + 1) % 3 == 0)
        m.addSeparator();
}
for(JCheckBoxMenuItem sfty : safety)
    s.add(sfty);
s.setMnemonic(KeyEvent.VK_A);
f.add(s);
f.setMnemonic(KeyEvent.VK_F);
for(int i = 0; i < file.length; i++) {
    file[i].addActionListener(ml);
    f.add(file[i]);
}
mb1.add(f);
mb1.add(m);
setJMenuBar(mb1);
t.setEditable(false);
add(t, BorderLayout.CENTER);
// Imposta il meccanismo per lo scambio dei menu:
b.addActionListener(new BL());
b.setMnemonic(KeyEvent.VK_S);
add(b, BorderLayout.NORTH);
for(JMenuItem oth : other)
    fooBar.add(oth);
fooBar.setMnemonic(KeyEvent.VK_B);
mb2.add(fooBar);
}
public static void main(String[] args) {
    run(new Menus(), 300, 200);
}
} ///:-
```



In questo programma le voci di menu sono inserite in array; poi questi array vengono elaborati, chiamando `add()` per ogni `JMenuItem`. Questo rende meno monotone le operazioni di aggiunta ed eliminazione dei menu.

Il programma crea due `JMenuBar` per dimostrare che le barre di menu possono essere scambiate mentre il programma è in funzione. Notate che una `JMenuBar` si compone di `JMenu`; ogni `JMenu` è composto di `JMenuItem`, `JCheckBoxMenuItem` o anche di altri `JMenu`, mediante i quali potete costruire sottome nu. Dopo aver assemblato una `JMenuBar` la si può installare nel programma chiamando il metodo `setJMenuBar()`. Osservate che quando viene premuto il pulsante, il programma chiama `getJMenuBar()` per controllare quale menu è installato in quel momento, poi installa la barra di menu alternativa.

Nel verificare la stringa, "Open" in questo caso, ricordate che l'ortografia e le maiuscole e minuscole sono fattori critici: Java non segnalera errori in caso di mancata corrispondenza con tale stringa. Tenete presente che questo tipo di verifica sulle stringhe è spesso fonte di errori di programmazione.

La selezione e la deselectazione delle voci di menu avvengono automaticamente. Il codice che gestisce i `JCheckBoxMenuItem` mostra due diverse tecniche per determinare ciò che è stato selezionato: la corrispondenza di stringhe, l'approccio meno sicuro, seppure molto utilizzato, e la corrispondenza sull'oggetto di destinazione dell'evento. Come vedete, è possibile utilizzare il metodo `getState()` per verificare lo stato; il metodo `setState()` consente invece di modificare lo stato di un `JCheckBoxMenuItem`.

Gli eventi per i menu sono in un certo senso contraddittori e potenziale fonte di confusione: `JMenuItem` si serve di `ActionListener` mentre `JCheckBoxMenuItem` utilizza `ItemListener`.

Gli oggetti `JMenu` possono anche supportare gli `ActionListener`, tuttavia in generale questo non è molto utile. Di norma collegherete i listener a ogni `JMenuItem`, `JCheckBoxMenuItem` o `JRadioButtonMenuItem`, sebbene l'esempio mostri che gli oggetti `ItemListener` e `ActionListener` sono connessi ai vari componenti di menu.

Swing supporta le abbreviazioni da tastiera, mediante le quali è possibile selezionare qualsiasi oggetto derivato da `AbstractButton` (pulsanti, voci di menu ecc.) utilizzando la tastiera anziché il mouse. Il meccanismo è piuttosto semplice: per i componenti `JMenuItem` potete servirvi del costruttore sovraccarico, che accetta, come secondo argomento, l'identificativo del tasto. Tuttavia la maggior parte degli oggetti `AbstractButton` non dispone di costruttori di questo tipo, pertanto l'approccio generico per creare le abbreviazioni da tastiera richiede il metodo `setMnemonic()`. Nell'esempio precedente sono state impostate alcune abbreviazioni da tastiera per il pulsante e alcune



voci di menu; le indicazioni dell'abbreviazione appaiono automaticamente sui componenti.

Il codice mostra anche l'utilizzo di `setActionCommand()`. Il suo impiego potrebbe apparirvi un po' strano, in quanto l'"*action command*" corrisponde esattamente all'etichetta sul componente del menu. Quindi, perché non utilizzare l'etichetta anziché questa stringa alternativa? I problemi nascono qualora si voglia internazionalizzare il programma: se traducete l'applicazione servendovi dell'"*action command*" potrete modificare solo l'etichetta nel menu, senza modificare il codice, evitando così il rischio di introdurre errori. Ricorrendo al metodo `setActionCommand()` si può impostare un "*action command*" che rimane costante, mentre l'etichetta del menu può essere cambiata. Tutto il codice si basa sugli action command, di conseguenza non è influenzato da modifiche alle etichette dei menu. Notate che in questo programma d'esempio l'*action command* non è utilizzato con tutti i componenti del menu; per questa ragione, i componenti per cui l'*action command* non viene utilizzato ne sono privi.

La maggior parte del lavoro si svolge nei listener. **BL** esegue lo scambio dei **JMenuBar**. In **ML** viene adottato l'approccio "indovina chi viene a cena?": una volta determinata la fonte dell'evento **ActionEvent**, questo viene convertito in un **JMenuItem**, da cui si ottiene la stringa *action command* che viene infine elaborata da una serie di istruzioni **if**.

Il listener **FL**, per quanto semplice, è in grado di gestire tutti i diversi sapori di gelato presenti nel menu. Questa tecnica è utile se la logica del menu è ragionevolmente limitata, ma in generale adotterete l'approccio di **Fool**, **BarL** e **BazL**, in cui ogni listener è collegato a un solo componente del menu: così facendo non occorre una logica supplementare e si può conoscere esattamente da quale elemento è stato chiamato il listener. Anche con la profusione di classi generata con questa tecnica, il codice relativo alle classi interne risulta più compatto e l'intero processo è meno soggetto a errori.

Come potete notare, il codice dei menu diventa rapidamente prolioso e poco leggibile. Questa è un'altra situazione in cui l'utilizzo di un ambiente di sviluppo visuale rappresenta la soluzione ideale: un buon IDE vi permetterà di gestire i menu al meglio.

**Esercizio 19 (3)** Modificate **Menus.java** per utilizzare menu con pulsanti radio anziché con caselle di scelta.

**Esercizio 20 (6)** Create un programma che suddivida un file di testo nelle singole parole che vi sono contenute: utilizzate queste parole per creare menu e sottomenu.



### **Menu pop-up**

Il modo più diretto per implementare un oggetto **JPopupMenu** consiste nel creare una classe interna che estenda **MouseListener**, aggiungendo poi un oggetto di questa classe a ogni componente cui si desidera assegnare il comportamento "pop-up" o "a comparsa":

```
//: gui/Popup.java
// Creazione di menu popup con Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Popup extends JFrame {
    private JPopupMenu popup = new JPopupMenu();
    private JTextField t = new JTextField(10);
    public Popup() {
        setLayout(new FlowLayout());
        add(t);
        ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                t.setText(((JMenuItem)e.getSource()).getText());
            }
        };
        JMenuItem m = new JMenuItem("Hither");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Yon");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Afar");
        m.addActionListener(al);
        popup.add(m);
        popup.addSeparator();
        m = new JMenuItem("Stay Here");
        m.addActionListener(al);
    }
}
```



```
popup.add(m);
PopupListener pl = new PopupListener();
addMouseListener(pl);
t.addMouseListener(pl);
}
class PopupListener extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        maybeShowPopup(e);
    }
    public void mouseReleased(MouseEvent e) {
        maybeShowPopup(e);
    }
    private void maybeShowPopup(MouseEvent e) {
        if(e.isPopupTrigger())
            popup.show(e.getComponent(), e.getX(), e.getY());
    }
}
public static void main(String[] args) {
    run(new Popup(), 300, 200);
}
} //:-~
```

Lo stesso **ActionListener** viene aggiunto a ogni **JMenuItem**, e preleva il testo presente nell'etichetta del menu per inserirlo nella casella di testo **JTextField**.

### Diseño

Una buona struttura GUI dovrebbe consentire di tracciare disegni in modo ragionevolmente facile: così è, in effetti, con la libreria Swing. Il problema con qualsiasi esempio di disegno è che i calcoli che determinano la posizione degli oggetti sono in genere molto più complessi delle chiamate alle routine di disegno, e che questi calcoli sono spesso mescolati con le chiamate di disegno, pertanto l'interfaccia potrebbe apparire più complessa di quanto non sia in realtà.

Per semplicità considerate il problema di rappresentare i dati a video: in questo caso i dati saranno forniti dal metodo nativo **Math.sin()**, che produce una funzione seno. Per rendere le cose più interessanti e dimostrare la facilità di utiliz-



zo dei componenti di Swing, nella parte inferiore del form viene posizionato un cursore scorrevole che gestisce dinamicamente il numero di cicli dell'onda di seno che verranno visualizzati. Inoltre, ridimensionando la finestra l'onda di seno si adatta alle nuove dimensioni.

Sebbene qualsiasi **JComponent** sia "disegnabile" e pertanto utilizzabile come supporto, per usufruire di una semplice superficie di disegno in genere si utilizza un **JPanel**. L'unico metodo da sovrascrivere è **paintComponent()**, che viene chiamato ogniqualvolta il componente deve essere ridisegnato: in ogni caso non dovete preoccuparvi di questo dettaglio, perché questa operazione è gestita da Swing. Quando viene chiamato **paintComponent()**, questo metodo riceve un oggetto **Graphics**: tale oggetto può essere poi utilizzato per disegnare o colorare la superficie.

Nell'esempio che segue tutta l'"intelligenza" relativa al disegno è contenuta nella classe **SineDraw**; la classe **SineWave** si limita a configurare il programma e il cursore scorrevole. All'interno di **SineDraw** viene utilizzato il metodo **setCycles()** per consentire a un altro componente, in questo caso il cursore scorrevole, di controllare il numero di cicli.

```
//: gui/SineWave.java
// Come disegnare con Swing utilizzando un cursore JSlider.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

class SineDraw extends JPanel {
    private static final int SCALEFACTOR = 200;
    private int cycles;
    private int points;
    private double[] sines;
    private int[] pts;
    public SineDraw() { setCycles(5); }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int maxWidth = getWidth();
        double hstep = (double)maxWidth / (double)points;
        int maxHeight = getHeight();
        pts = new int[points];
        for (int i = 0; i < points; i++) {
            pts[i] = (int)(Math.sin((double)i * 2 * Math.PI / cycles) * (maxHeight / 2) + maxHeight / 2);
        }
        g.drawLine(0, maxHeight, maxWidth, maxHeight);
        for (int i = 0; i < points; i++) {
            g.drawLine(i * hstep, maxHeight, i * hstep, pts[i]);
        }
    }
    public void setCycles(int cycles) {
        this.cycles = cycles;
        repaint();
    }
}
```



```
for(int i = 0; i < points; i++)  
    pts[i] =  
        (int)(sines[i] * maxHeight/2 * .95 + maxHeight/2);  
    g.setColor(Color.RED);  
    for(int i = 1; i < points; i++) {  
        int x1 = (int)((i - 1) * hstep);  
        int x2 = (int)(i * hstep);  
        int y1 = pts[i-1];  
        int y2 = pts[i];  
        g.drawLine(x1, y1, x2, y2);  
    }  
}  
public void setCycles(int newCycles) {  
    cycles = newCycles;  
    points = SCALEFACTOR * cycles * 2;  
    sines = new double[points];  
    for(int i = 0; i < points; i++) {  
        double radians = (Math.PI / SCALEFACTOR) * i;  
        sines[i] = Math.sin(radians);  
    }  
    repaint();  
}  
}  
  
public class SineWave extends JFrame {  
    private SineDraw sines = new SineDraw();  
    private JSlider adjustCycles = new JSlider(1, 30, 5);  
    public SineWave() {  
        add(sines);  
        adjustCycles.addChangeListener(new ChangeListener() {  
            public void stateChanged(ChangeEvent e) {  
                sines.setCycles(  
                    ((JSlider)e.getSource()).getValue());  
            }  
        });  
        add(BorderLayout.SOUTH, adjustCycles);  
    }  
}
```



```
public static void main(String[] args) {
    run(new SineWave(), 700, 400);
}
} //:~
```

Tutti i campi e gli array sono richiesti per il calcolo dei punti dell'onda di seno; **cycles** indica il numero di onde di seno complete desiderate, **points** il numero di punti che saranno rappresentati graficamente, **sines** i valori della funzione seno e **pts** le coordinate *y* dei punti da tracciare su **JPanel**. Il metodo **setCycles()** crea gli array in base al numero di punti necessari e popola di numeri l'array **sines**. Chiamando il metodo **repaint()**, **setCycles()** provoca la chiamata di **paintComponent()** per eseguire le rimanenti operazioni di calcolo e di disegno.

La prima cosa che dovete fare quando sovrascrivete **paintComponent()** è chiamare la versione della classe di base del metodo. A quel punto sarete liberi di procedere in qualunque modo riteniate opportuno; di norma vi servirete dei metodi **Graphics** disponibili per **java.awt.Graphics** per tracciare e colorare i pixel su **JPanel**: consultate la documentazione JDK disponibile all'indirizzo <http://java.sun.com> per conoscere l'elenco di questi metodi. In questo caso specifico, potete vedere che quasi tutto il codice è coinvolto nell'esecuzione dei calcoli; le uniche due chiamate di metodo che gestiscono lo schermo sono **setColor()** e **drawLine()**. Probabilmente vi troverete in una situazione analoga quando creerete il vostro programma che visualizza i dati grafici; dedicherete la maggior parte del codice a calcolare ciò che volete disegnare, mentre il processo di disegno vero e proprio sarà abbastanza semplice.

Quando l'autore ha creato questo programma, ha dedicato quasi tutto il tempo alla visualizzazione dell'onda di seno. Fatto questo, ha ritenuto utile modificare dinamicamente il numero di cicli. Le esperienze di Eckel con altri linguaggi lo avevano reso alquanto riluttante nei confronti di questo tipo di programmazione, che si è però rivelata la parte più facile del progetto. È bastato creare un cursore **JSlider** e includerlo in un **JFrame**: gli argomenti di **JSlider** sono rispettivamente il valore della posizione all'estrema sinistra, quello all'estrema destra e il valore di partenza, tuttavia sono disponibili altri costruttori.

A quel punto, esaminando la documentazione JDK l'autore ha notato che l'unico listener era **addChangeListener**, che si attiva ogniqualvolta il cursore scorrevole viene spostato in misura sufficiente da produrre un valore diverso. L'unico metodo utilizzabile a questo scopo era **stateChanged()**, che fornisce un oggetto **ChangeEvent**: questo ha permesso all'autore di risalire all'origine del cambiamento e trovare il nuovo valore. La chiamata al metodo **setCycles()** de-



gli oggetti `sines` consente l'inclusione del nuovo valore nel **JPanel** e il ridisegno di quest'ultimo.

Generalmente, la maggior parte dei problemi con Swing è risolvibile con un approccio analogo e troverete che è abbastanza semplice, anche se non avete mai utilizzato prima il componente in questione.

Se il problema da risolvere è più complesso, esistono alternative di disegno più specializzate, tra cui componenti JavaBeans di altri produttori e le API 2D di Java. Queste soluzioni vanno oltre gli obiettivi di questo volume, tuttavia dovreste tenerle in considerazione qualora il vostro codice diventasse troppo oneroso da gestire.

**Esercizio 21 (5)** Modificate `SineWave.java` per trasformare `SineDraw` in un JavaBean, aggiungendo i metodi “getter” e “setter”.

**Esercizio 22 (7)** Servendovi di `SwingConsole` create un'applicazione, che dovrà comprendere tre cursori, uno per ognuno dei valori RGB (rosso, verde e blu) in `java.awt.Color`. Il colore risultante dalla combinazione dei tre cursori dovrà essere visualizzato in un **JPanel**. Includete anche campi di testo non editabili che mostrino i valori correnti RGB.

**Esercizio 23 (8)** Basandovi su `SineWave.java`, create un programma che visualizzi un quadrato rotante sullo schermo. La velocità di rotazione sarà gestibile per mezzo di un cursore, e un secondo cursore dovrà gestire le dimensioni del quadrato.

**Esercizio 24 (7)** Forse ricorderete quel giocattolo in cui due manopole che gestiscono lo scorrimento di un cursore, in senso verticale e orizzontale, permettono di tracciare figure su uno schermo cerato. Create una variante di questo giocattolo, usando `SineWave.java` come codice di partenza. Come “manopole” utilizzate due cursori e aggiungete un pulsante per ripulire il contenuto dello schermo.

**Esercizio 25 (8)** Partendo da `SineWave.java` create un'applicazione che utilizzi la classe `SwingConsole`, per tracciare un'onda di seno animata che sembri scorrere oltre i limiti della finestra, come avviene in un oscilloscopio. Per muovere l'animazione utilizzate `java.util.Timer`, e un controllo `javax.swing.JSlider` per impostare la velocità di animazione.

**Esercizio 26 (5)** Modificate l'esercizio precedente in modo da creare diversi pannelli che visualizzino le onde di seno, all'interno dell'applicazione. Il numero di pannelli dovrà essere gestibile da riga di comando.

**Esercizio 27 (5)** Modificate l'esercizio 25 per utilizzare la classe `javax.swing.Timer` che controlli l'animazione. Notate la differenza tra questa classe e `java.util.Timer`.



**Esercizio 28 (7)** Create una classe per gestire i dadi da gioco, senza interfaccia grafica, poi generate cinque dadi e lanciateli ripetutamente. Quindi tracciate la curva che mostra la somma dei punti ottenuti per ogni lancia, e mostrate che la curva si evolve dinamicamente a ogni lancia di dadi.

### ***Finestre di dialogo***

Una finestra di dialogo viene generata da un'altra finestra dell'applicazione. Le finestre di dialogo sono comunemente utilizzate negli ambienti di sviluppo grafici per gestire situazioni specifiche senza riempire la finestra principale dell'applicazione con eccessivi dettagli. Per creare una finestra di dialogo è necessario estendere la classe **JDialog**, che è un tipo di **Window**, analogo a un **JFrame**. Un **JDialog** dispone di un layout manager (**BorderLayout** è quello predefinito) e può integrare alcuni listener di eventi addizionali. Considerate il semplice esempio seguente:

```
//: gui/Dialogs.java
// Creazione e utilizzo delle finestre di dialogo.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

class MyDialog extends JDialog {
    public MyDialog(JFrame parent) {
        super(parent, "My dialog", true);
        setLayout(new FlowLayout());
        add(new JLabel("Here is my dialog"));
        JButton ok = new JButton("OK");
        ok.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dispose(); // Chiude la finestra di dialogo
            }
        });
        add(ok);
        setSize(200,125);
    }
}
```



```

public class Dialogs extends JFrame {
    private JButton b1 = new JButton("Dialog Box");
    private MyDialog dlg = new MyDialog(null);
    public Dialogs() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dlg.setVisible(true);
            }
        });
        add(b1);
    }
    public static void main(String[] args) {
        run(new Dialogs(), 180, 75);
    }
} //:-

```

Una volta che l'oggetto **JDialog** è stato creato, è necessario chiamare **setVisible(true)** per visualizzarlo e attivarlo; quando la finestra di dialogo viene chiusa, occorre liberare le risorse che sono state utilizzate, chiamando il metodo **dispose()**.

L'esempio seguente è più articolato: la finestra di dialogo si compone di una griglia, gestita da **GridLayout**, composta di pulsanti speciali definiti come classe **ToeButton**. Questo particolare pulsante disegna un frame attorno a sé, e nella parte centrale può visualizzare, in funzione del proprio stato, uno spazio, una "X" o una "O", i simboli del classico gioco del Tris. La condizione iniziale corrisponde allo spazio vuoto, poi, in base al giocatore che detiene il turno, si trasforma in una "X" o in una "O". Per fornire una variazione rispetto al normale gioco del tris, ogniqualvolta si fa clic su uno dei pulsanti questi si trasformano in "X" o "O", in modo alterno. Le dimensioni di questa finestra di dialogo, contenente lo schema del tris, possono essere impostate in un numero arbitrario di righe e colonne, semplicemente modificando i relativi valori nella finestra principale.

```

//: gui/TicTacToe.java
// Finestre di dialogo e creazione di componenti
// personalizzati.
import javax.swing.*;
import java.awt.*;

```



```
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class TicTacToe extends JFrame {
    private JTextField
        rows = new JTextField("3"),
        cols = new JTextField("3");
    private enum State { BLANK, XX, OO }
    static class ToeDialog extends JDialog {
        private State turn = State.XX; // Inizia con il turno X
        ToeDialog(int cellsWide, int cellsHigh) {
            setTitle("The game itself");
            setLayout(new GridLayout(cellsWide, cellsHigh));
            for(int i = 0; i < cellsWide * cellsHigh; i++)
                add(new ToeButton());
            setSize(cellsWide * 50, cellsHigh * 50);
            setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        }
        class ToeButton extends JPanel {
            private State state = State.BLANK;
            public ToeButton() { addMouseListener(new ML()); }
            public void paintComponent(Graphics g) {
                super.paintComponent(g);
                int
                    x1 = 0, y1 = 0,
                    x2 = getSize().width - 1,
                    y2 = getSize().height - 1;
                g.drawRect(x1, y1, x2, y2);
                x1 = x2/4;
                y1 = y2/4;
                int wide = x2/2, high = y2/2;
                if(state == State.XX) {
                    g.drawLine(x1, y1, x1 + wide, y1 + high);
                    g.drawLine(x1, y1 + high, x1 + wide, y1);
                }
                if(state == State.OO)
```



```
        g.drawOval(x1, y1, x1 + wide/2, y1 + high/2);
    }
}

class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        if(state == State.BLANK) {
            state = turn;
            turn =
                (turn == State.XX ? State.OO : State.XX);
        }
        else
            state =
                (state == State.XX ? State.OO : State.XX);
        repaint();
    }
}
}

class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JDialog d = new ToeDialog(
            new Integer(rows.getText()),
            new Integer(cols.getText()));
        d.setVisible(true);
    }
}
public TicTacToe() {
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(2,2));
    p.add(new JLabel("Rows", JLabel.CENTER));
    p.add(rows);
    p.add(new JLabel("Columns", JLabel.CENTER));
    p.add(cols);
    add(p, BorderLayout.NORTH);
    JButton b = new JButton("go");
    b.addActionListener(new BL());
    add(b, BorderLayout.SOUTH);
}
```



```
public static void main(String[] args) {  
    run(new TicTacToe(), 200, 200);  
}  
} //:~
```

Il metodo **paintComponent()** traccia il riquadro attorno al pulsante e i simboli “X” o “O”: questa parte del programma esegue numerosi calcoli, ma è piuttosto semplice.

Il clic del mouse viene intercettato da **MouseListener**, che verifica se il pulsante visualizza un simbolo. Se non è visualizzato alcun simbolo, alla finestra principale viene richiesto il “turno”, stabilendo così la condizione del componente **ToeButton**; tramite il meccanismo della classe interna, **ToeButton** raggiunge nuovamente la classe principale e modifica il turno. Se invece il pulsante visualizza una “X” o una “O”, tale simbolo viene invertito: in questi algoritmi potete vedere l'utilizzo del pratico operatore ternario **if-else**, descritto nel Volume 1, Capitolo 3. Dopo ogni cambiamento di stato il componente **ToeButton** viene nuovamente visualizzato.

Il costruttore di **ToeDialog** è piuttosto semplice: aggiunge in un **GridLayout** il numero di pulsanti richiesto, quindi li ridimensiona alla dimensione di 50 pixel per lato.

**TicTacToe** imposta l'intera applicazione creando i due campi **JTextField**, nei quali vanno inseriti i numeri di righe e colonne, e il pulsante “go” con il relativo **ActionListener**. Quando premete il pulsante vengono prelevati i dati contenuti nei **JTextField**: trattandosi di **String**, sono convertiti in **int** servendosi del costruttore di **Integer** che accetta un argomento **String**.

### Finestre di dialogo per i file

Molti sistemi operativi mettono a disposizione finestre di dialogo incorporate per gestire la selezione di oggetti quali i caratteri, i colori, le stampanti ecc. Presoché tutti i sistemi operativi supportano l'apertura e il salvataggio dei file: il componente Java **JFileChooser** incapsula queste funzionalità rendendole facilmente accessibili. L'applicazione mostrata di seguito utilizza due tipi di finestre di dialogo **JFileChooser**, una per l'apertura e una per il salvataggio dei file. La maggior parte del codice dovrebbe esservi ormai familiare; tutte le attività importanti avvengono negli action listener che gestiscono il clic sui due pulsanti:

```
//: gui/FileChooserTest.java  
// Dimostrazione delle finestre di dialogo per i file.  
import javax.swing.*;
```



```
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class FileChooserTest extends JFrame {
    private JTextField
        fileName = new JTextField(),
        dir = new JTextField();
    private JButton
        open = new JButton("Open"),
        save = new JButton("Save");
    public FileChooserTest() {
        JPanel p = new JPanel();
        open.addActionListener(new OpenL());
        p.add(open);
        save.addActionListener(new SaveL());
        p.add(save);
        add(p, BorderLayout.SOUTH);
        dir.setEditable(false);
        fileName.setEditable(false);
        p = new JPanel();
        p.setLayout(new GridLayout(2,1));
        p.add(fileName);
        p.add(dir);
        add(p, BorderLayout.NORTH);
    }
    class OpenL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JFileChooser c = new JFileChooser();
            // Finestra di dialogo "Apri";
            int rVal = c.showOpenDialog(FileChooserTest.this);
            if(rVal == JFileChooser.APPROVE_OPTION) {
                fileName.setText(c.getSelectedFile().getName());
                dir.setText(c.getCurrentDirectory().toString());
            }
            if(rVal == JFileChooser.CANCEL_OPTION) {
                fileName.setText("You pressed cancel");
            }
        }
    }
}
```



```
        dir.setText("");
    }
}
}

class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JFileChooser c = new JFileChooser();
        // Finestra di dialogo "Salva":
        int rVal = c.showSaveDialog(FileChooserTest.this);
        if(rVal == JFileChooser.APPROVE_OPTION) {
            fileName.setText(c.getSelectedFile().getName());
            dir.setText(c.getCurrentDirectory().toString());
        }
        if(rVal == JFileChooser.CANCEL_OPTION) {
            fileName.setText("You pressed cancel");
            dir.setText("");
        }
    }
}

public static void main(String[] args) {
    run(new FileChooserTest(), 250, 150);
}
} ///:~
```

Tenete presente che a **JFileChooser** possono essere applicate numerose varianti, per esempio impostando filtri sui nomi dei file.

Per aprire una finestra di dialogo “Open” dovete chiamare il metodo **showOpenDialog()**; per la finestra di dialogo “Save” vi servirete invece di **showSaveDialog()**. Tenete presente che questi metodi non ritornano fino alla chiusura della finestra di dialogo. L’oggetto **JFileChooser** continua a esistere anche dopo la chiusura della finestra di dialogo, quindi potete continuare a leggerne i dati. I metodi **getSelectedFile()** e **getCurrentDirectory()** vi consentono di consultare i risultati dell’operazione: se la chiamata a questi metodi restituisce **null** significa che l’utente ha annullato l’operazione.

**Esercizio 29** (3) Nella documentazione JDK di **javax.swing** consultate le informazioni per la classe **JColorChooser**, poi scrivete un programma con un pulsante che apre la finestra di dialogo per la selezione dei colori.



## ***HTML nei componenti Swing***

Qualsiasi componente che accetta testo può anche ricevere codice HTML, che sarà formattato secondo le regole di questo linguaggio di marcatura. È quindi molto semplice aggiungere testo formattato a un componente Swing, come nell'esempio seguente:

```
//: gui/HTMLButton.java
// Come inserire del testo HTML nei componenti Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class HTMLButton extends JFrame {
    private JButton b = new JButton(
        "<html><b><font size=+2>" +
        "<center>Hello!<br><i>Press me now!" );
    public HTMLButton() {
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                add(new JLabel("<html>" +
                    "<i><font size=+4>Kapow!" ));
                // Ridisegna la pagina per tener conto della nuova
                // etichetta:
                validate();
            }
        });
        setLayout(new FlowLayout());
        add(b);
    }
    public static void main(String[] args) {
        run(new HTMLButton(), 200, 500);
    }
} ///:~
```

La porzione di testo formattato deve iniziare con il tag “`<html>`” e può includere tutti i normali tag di formattazione HTML. Tenete presente che i tag di chiusura non sono obbligatori.



In questo esempio **ActionListener** aggiunge al modulo una nuova **JLabel**, contenente anch'essa una porzione di testo HTML. Questa etichetta tuttavia non viene aggiunta durante la costruzione, di conseguenza dovete chiamare il metodo **validate()** del contenitore per forzare una nuova visualizzazione dei componenti, e quindi della nuova etichetta.

Potete utilizzare il testo HTML anche in **JTabbedPane**, **JMenuItem**, **JToolTip**, **JRadioButton** e **JCheckBox**.

**Esercizio 30** (3) Scrivete un programma che mostra l'utilizzo del testo HTML con gli oggetti **JTabbedPane**, **JMenuItem**, **JToolTip**, **JRadioButton** e **JCheckBox**.

### **Cursori scorrevoli e barre di avanzamento**

Il cursore scorrevole, che avete già utilizzato in **SineWave.java**, vi permette di eseguire l'input dei dati spostando un cursore. Questo meccanismo in alcune situazioni è decisamente intuitivo: si pensi per esempio ai controlli del volume. Una barra di avanzamento mostra i dati con un tipico aspetto da "vuoto" a " pieno", fornendo all'utente una prospettiva dello stato di completamento di un'operazione. L'esempio preferito dell'autore consiste nel collegare direttamente la barra di avanzamento al cursore scorrevole: intervenendo sul cursore scorrevole, la barra si sposta proporzionalmente. L'esempio seguente illustra anche il componente **ProgressMonitor**, una finestra pop-up arricchita di maggiori funzionalità:

```
//: gui/Progress.java
// Utilizzo di cursori, barre d'avanzamento e progress
// monitor.

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Progress extends JFrame {
    private JProgressBar pb = new JProgressBar();
    private ProgressMonitor pm = new ProgressMonitor(
        this, "Monitoring Progress", "Test", 0, 100);
    private JSeparator sb =
        new JSeparator(JSeparator.HORIZONTAL, 0, 100, 60);
```



```
public Progress() {
    setLayout(new GridLayout(2,1));
    add(pb);
    pm.setProgress(0);
    pm.setMillisToPopup(1000);
    sb.setValue(0);
    sb.setPaintTicks(true);
    sb.setMajorTickSpacing(20);
    sb.setMinorTickSpacing(5);
    sb.setBorder(new TitledBorder("Slide Me"));
    pb.setModel(sb.getModel()); // Condivisione del modello
    add(sb);
    sb.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
            pm.setProgress(sb.getValue());
        }
    });
}
public static void main(String[] args) {
    run(new Progress(), 300, 200);
}
} ///:-~
```

La chiave che collega il cursore scorrevole alla barra d'avanzamento è la condivisione del loro modello, nella riga:

```
pb.setModel(sb.getModel());
```

Naturalmente potreste gestire entrambi questi controlli anche per mezzo di un listener, ma l'utilizzo del modello risulta più pratico nelle situazioni più semplici. Il componente **ProgressMonitor** non possiede un modello, quindi è necessario impostare un listener. Notate che il **ProgressMonitor** si sposta soltanto in avanti, chiudendo automaticamente la propria finestra una volta raggiunta l'estremità del cursore scorrevole.

Il componente **JProgressBar** è piuttosto semplice, mentre **JSlider** possiede numerose opzioni quali l'orientamento e la possibilità di modificare la suddivisione della scala graduata. Notate anche quanto sia semplice aggiungere un bordo con un titolo.



**Esercizio 31** (8) Create un “indicatore asintotico di progresso” che avanzi tanto più lentamente quanto più si avvicina al termine. Simulate anche un comportamento casuale, così che di tanto in tanto l’indicatore sembri accelerare.

**Esercizio 32** (6) Modificate **Progress.java** in modo che, invece di condividere i modelli, si serva di un listener per collegare il cursore scorrevole e la barra di avanzamento.

### **Modificare l’aspetto dell’interfaccia**

L’approccio “pluggable look & feel” fa sì che il vostro programma emuli l’aspetto di vari ambienti operativi, anche in modo dinamico, mentre il programma è in esecuzione. Di norma sceglierete l’aspetto “cross platform”, rappresentato dal modello “metal” della libreria Swing, oppure quello tipico del sistema operativo che ospita il vostro programma. Quest’ultima è quasi sempre la scelta migliore, poiché evita di confondere l’utente.

Il codice di selezione è piuttosto semplice e deve essere eseguito *prima* di creare tutti i componenti grafici; infatti i componenti vengono creati in base all’aspetto selezionato, e una volta creati non cambiano al variare dell’aspetto globale dell’ambiente grafico: si tratta di un processo complesso, che generalmente è trattato soltanto in volumi specificamente dedicati a Swing.

Per impostare l’aspetto “cross platform” (“metal”) non dovete eseguire alcuna operazione, in quanto si tratta del comportamento predefinito. Se preferite invece l’aspetto tipico del sistema operativo in uso, dovrete inserire il codice seguente, di solito all’inizio del metodo **main()** ma, come si è detto, prima dell’aggiunta di qualsiasi componente:

```
try {
    UIManager.setLookAndFeel(
        UIManager.getSystemLookAndFeelClassName());
} catch(Exception e) {
    throw new RuntimeException(e);
}
```

Non occorre che inseriate alcunché nell’istruzione **catch**, perché qualora il tentativo di impostare un particolare aspetto fallisca, **UIManager** utilizzerà automaticamente quello predefinito; tuttavia, in fase di debug, l’intercettazione dell’eccezione potrebbe risultarvi utile.



Ecco un programma che imposta l'aspetto in base al parametro fornito come argomento da riga di comando, adattando l'aspetto dei vari componenti ai diversi casi:

```
//: gui/LookAndFeel.java
// Come selezionare diversi aspetti.
// {Args: motif}
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class LookAndFeel extends JFrame {
    private String[] choices =
        "Eeny Meeny Minnie Mickey Moe Larry Curly".split(" ");
    private Component[] samples = {
        new JButton("JButton"),
        new JTextField("JTextField"),
        new JLabel("JLabel"),
        new JCheckBox("JCheckBox"),
        new JRadioButton("Radio"),
        new JComboBox(choices),
        new JList(choices),
    };
    public LookAndFeel() {
        super("Look And Feel");
        setLayout(new FlowLayout());
        for(Component component : samples)
            add(component);
    }
    private static void usageError() {
        System.out.println(
            "Usage:LookAndFeel [cross|system|motif]");
        System.exit(1);
    }
    public static void main(String[] args) {
        if(args.length == 0) usageError();
        if(args[0].equals("cross")) {
```



```
try {
    UIManager.setLookAndFeel(UIManager.
        getCrossPlatformLookAndFeelClassName());
} catch(Exception e) {
    e.printStackTrace();
}
} else if(args[0].equals("system")) {
    try {
        UIManager.setLookAndFeel(UIManager.
            getSystemLookAndFeelClassName());
    } catch(Exception e) {
        e.printStackTrace();
    }
} else if(args[0].equals("motif")) {
    try {
        UIManager.setLookAndFeel("com.sun.java." +
            "swing.plaf.motif.MotifLookAndFeel");
    } catch(Exception e) {
        e.printStackTrace();
    }
} else usageError();
// Notate che l'aspetto deve essere impostato
// prima della creazione dei componenti.
run(new LookAndFeel(), 300, 300);
}
} //:~
```

Nella selezione di **MotifLookAndFeel** notate che è possibile specificare un aspetto sotto forma di stringa. L'aspetto “metal” è l'unico utilizzabile su qualsiasi piattaforma, poiché quelli per Windows e Macintosh sono utilizzabili soltanto sulle rispettive piattaforme, chiamando il metodo **getSystemLookAndFeelClassName()** dalla piattaforma “ospite”.

Potete anche creare un pacchetto “look & feel” personalizzato: questa possibilità è interessante, per esempio, se state sviluppando per un'azienda che desidera un aspetto distintivo. Si tratta comunque di un'operazione che va oltre la portata di questo manuale: del resto, si rivela anche oltre la portata di molti manuali dedicati esplicitamente a Swing!



## Strutture ad albero, tavole e appunti

Potete trovare una breve introduzione e gli esempi per questi argomenti nei supplementi in linea disponibili all'indirizzo [www.mindview.net](http://www.mindview.net).

## JNLP e Java web Start

Per soddisfare esigenze di sicurezza è possibile *firmare* un applet, ricorrendo alla tecnica illustrata nel supplemento in linea per questo capitolo, che potete procurarvi all'indirizzo [www.mindview.net](http://www.mindview.net). Gli applet firmati sono potenti e possono sostituire efficacemente un'applicazione, ma devono essere eseguiti all'interno di un browser web. Questo implica non soltanto l'onere rappresentato dall'esecuzione del browser sul sistema client, ma anche alcune limitazioni relative all'interfaccia utente, che per gli applet è limitata e talvolta confusa, considerando che il browser web possiede un proprio insieme di menu e barre di strumenti che compare al di sopra dell'applet.<sup>7</sup>

JNLP (*Java Network Launch Protocol*) risolve questo problema senza sacrificare i vantaggi degli applet. Con un'applicazione JNLP, potete trasferire e installare un'applicazione Java autonoma sul sistema client. Questa applicazione può essere eseguita da riga di comando, da un'icona sul desktop, da un apposito gestore installato con la vostra implementazione di JNLP o direttamente dal sito web che la ospita.

Un'applicazione JNLP può scaricare dinamicamente da Internet le risorse necessarie durante l'esecuzione, oltre a verificare automaticamente la versione dell'applicazione quando l'utente è collegato a Internet. In tal modo, si ottengono tutti i vantaggi di un applet connessi ai benefici di un'applicazione autonoma.

Come gli applet, le applicazioni JNLP devono essere gestite con grande attenzione dal sistema client, e per questo motivo le applicazioni JNLP sono soggette alle stesse limitazioni di sicurezza della *sandbox* degli applet. Come gli applet, possono essere distribuite tramite file JAR, dando all'utente la possibilità di fidarsi del "firmatario". A differenza degli applet, però, se le applicazioni JNLP sono distribuite in un file JAR non firmato possono comunque accedere a determinate risorse del sistema client, tramite i servizi delle API JNLP; l'utente dovrà tuttavia autorizzare tali accessi durante l'esecuzione dell'applicazione.

JNLP descrive un protocollo, quindi vi occorrerà un'implementazione per utilizzarlo. Java web Start, o JAWS, è l'implementazione di riferimento ufficiale che Sun offre gratuitamente e distribuisce come componente di Java SE5. Se volete servirvi di questa tecnologia per lo sviluppo di applicazioni,

---

7. Questa sezione è stata sviluppata da Jeremy Meyer.



dovrete accertarvi che il file JAR **javaws.jar** sia presente nel vostro CLASSPATH; la soluzione più semplice consiste nell'aggiungere **javaws.jar** al CLASSPATH in **jre/lib**. Se state distribuendo un'applicazione di JNLP da un server web, dovrete accertarvi che il server riconosca il tipo **MIME application/x-Java-jnlp-file**: questa opzione è già configurata nelle versioni recenti del server Tomcat (<http://jakarta.apache.org/tomcat>), ma se utilizzate un server diverso dovrete consultare il manuale del prodotto.

Non è difficile creare un'applicazione JNLP: vi basterà realizzare un'applicazione standard archiviata in un file JAR, poi fornire un “file di lancio”, ossia un semplice file XML che fornisce al sistema client le informazioni necessarie per scaricare ed eseguire l'applicazione. Se scegliete di non firmare il vostro file JAR, dovrete utilizzare i servizi forniti dalle API JNLP per ogni tipo di risorsa a cui desiderate accedere sul sistema client.

Di seguito è mostrata una variante dell'esempio **FileChooserTest.java**, che si serve dei servizi JNLP per aprire la finestra di dialogo “File”; questa classe può essere distribuita come applicazione JNLP in un file JAR non firmato.

```
//: gui/jnlp/JnlpFileChooser.java
// Apre i file su un sistema locale tramite JNLP.
// {Richiede javax.jnlp.FileOpenService;
// E' necessario che javaws.jar sia elencato nel classpath}
// Per creare il file jnlpfilechooser.jar eseguite:
// cd ..
// cd ..
// jar cvf gui/jnlp/jnlpfilechooser.jar gui/jnlp/*.class
package gui.jnlp;
import javax.jnlp.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class JnlpFileChooser extends JFrame {
    private JTextField fileName = new JTextField();
    private JButton
        open = new JButton("Open"),
        save = new JButton("Save");
    private JEditorPane ep = new JEditorPane();
```



```
private JScrollPane jsp = new JScrollPane();
private FileContents fileContents;
public JnlpFileChooser() {
    JPanel p = new JPanel();
    open.addActionListener(new OpenL());
    p.add(open);
    save.addActionListener(new SaveL());
    p.add(save);
    jsp.setViewportView(ep);
    add(jsp, BorderLayout.CENTER);
    add(p, BorderLayout.SOUTH);
    fileName.setEditable(false);
    p = new JPanel();
    p.setLayout(new GridLayout(2,1));
    p.add(fileName);
    add(p, BorderLayout.NORTH);
    ep.setContentType("text");
    save.setEnabled(false);
}
class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        FileOpenService fs = null;
        try {
            fs = (FileOpenService)ServiceManager.lookup(
                "javax.jnlp.FileOpenService");
        } catch(UnavailableServiceException use) {
            throw new RuntimeException(use);
        }
        if(fs != null) {
            try {
                fileContents = fs.openFileDialog(".", 
                    new String[]{"txt", "="});
                if(fileContents == null)
                    return;
                fileName.setText(fileContents.getName());
                ep.read(fileContents.getInputStream(), null);
            } catch(Exception exc) {
```



```
        throw new RuntimeException(exc);
    }
    save.setEnabled(true);
}
}
}

class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        FileSaveService fs = null;
        try {
            fs = (FileSaveService)ServiceManager.lookup(
                "javax.jnlp.FileSaveService");
        } catch(UnavailableServiceException use) {
            throw new RuntimeException(use);
        }
        if(fs != null) {
            try {
                fileContents = fs.saveFileDialog(".", 
                    new String[]{"txt"}, 
                    new ByteArrayInputStream(
                        ep.getText().getBytes(),
                        fileContents.getName()));
                if(fileContents == null)
                    return;
                fileName.setText(fileContents.getName());
            } catch(Exception exc) {
                throw new RuntimeException(exc);
            }
        }
    }
}

public static void main(String[] args) {
    JnlpFileChooser fc = new JnlpFileChooser();
    fc.setSize(400, 300);
    fc.setVisible(true);
}
} //:-:
```



Noteate che le classi **FileOpenService** e **FileSaveService** sono importate dal package **javax.jnlp**, e il codice non fa direttamente riferimento alla finestra di dialogo **JFileChooser**. I due servizi utilizzati devono essere richiesti per mezzo del metodo **ServiceManager.lookup()**: le risorse presenti sul sistema client possono essere raggiunte soltanto tramite gli oggetti restituiti da questo metodo. In questo caso i file sul filesystem client vengono letti e modificati mediante l'interfaccia **FileContent**, fornita da JNLP. Il tentativo di accedere direttamente alle risorse, utilizzando per esempio un oggetto **File** o **FileReader**, solleverebbe un'eccezione **SecurityException**, esattamente come accadrebbe con un applet non firmato. Se volete servirvi di queste classi senza essere limitati alle interfacce di servizio di JNLP, dovrete firmare il file JAR.

Il comando presente nei commenti di **JnlpFileChooser.java** produrrà il necessario file JAR. Di seguito è mostrato il file XML necessario per eseguire l'esempio precedente.

```
//:! gui/jnlp/filechooser.jnlp
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec = "1.0+"
codebase=" C:/AAA-TI34/code/gui/jnlp"
href="filechooser.jnlp">
<information>
<title>FileChooser demo application</title>
<vendor>Mindview Inc.</vendor>
<description>
  Jnlp File chooser Application
</description>
<description kind="short">
  Demonstrates opening, reading and writing a text file
</description>
<icon href="mindview.gif"/>
<offline-allowed/>
</information>
<resources>
<j2se version="1.3+"
  href="http://java.sun.com/products/autodl/j2se"/>
<jar href="jnlpfilechooser.jar" download="eager"/>
</resources>
```



```
<application-desc  
    main-class="gui.jnlp.JnlpFileChooser"/>  
</jnlp>  
///:~
```

Troverete questo file di lancio nel file contenente il codice sorgente di questo manuale, scaricabile da [www.mindview.net](http://www.mindview.net), salvato come **filechooser.jnlp** e privo della prima e dell'ultima riga di commento, nella stessa directory del file JAR, che dovrete costruire nella stessa directory contenente il codice sorgente. Come potete vedere, è un semplice file XML con un tag **<jnlp>**; sono presenti alcuni sottoelementi, molti dei quali autoesplicativi.

L'attributo **spec** dell'elemento **jnlp** indica al sistema client la versione di JNLP necessaria per eseguire l'applicazione. L'attributo **codebase** segnala l'URL in cui possono essere reperiti il file di lancio e le risorse aggiuntive: in questo caso specifico indica una directory sul sistema corrente, una buona tecnica per testare l'applicazione. *Tenete presente che per eseguire il programma dovrete modificare questo percorso in modo che indichi la directory appropriata sul vostro computer.* L'attributo **href** deve specificare il nome del file di lancio **jnlp**. Il tag **information** dispone di alcuni sottoelementi che forniscono informazioni sull'applicazione. Questi dati sono utilizzati dalla console amministrativa di Java web Start o da programmi equivalenti, che installano l'applicazione JNLP e permettono all'utente di avviarla da riga di comando, di creare collegamenti e così via.

Il tag **resources** ha uno scopo simile al tag *applet* presente nei file HTML. Il sottoelemento **j2se** indica la versione di J2SE necessaria per eseguire l'applicazione, mentre **jar** specifica il nome del file JAR in cui è contenuta l'applicazione. L'elemento **jar** possiede un attributo **download**, che può assumere i valori "eager" o "lazy": tali valori indicano all'implementazione JNLP se occorre scaricare l'intero archivio prima di eseguire l'applicazione. L'attributo **application-desc** indica qual è la classe eseguibile, o *punto di entrata* (*entry point*), del file JAR. Un altro sottoelemento utile del tag **jnlp** è **security**, assente nel codice in esame, che ha l'aspetto seguente:

```
<security>  
    <all-permissions/>  
<security/>
```

Il tag **security** deve essere utilizzato quando l'applicazione viene distribuita in un file JAR firmato. Nell'esempio precedente non è necessario, perché le risorse locali sono utilizzate tramite i servizi di JNLP.

Vi sono altri tag disponibili, che troverete descritti in dettaglio nelle specifiche tecniche pubblicate da Sun, disponibili all'indirizzo <http://java.sun.com/products/javawebstart/download-spec.html>.

Per avviare il programma, occorre predisporre una pagina web contenente un collegamento ipertestuale per il download del file **.jnlp**. Ecco un esempio di file HTML adatto: la prima e l'ultima riga devono essere eliminate.

```
//:! gui/jnlp/filechooser.html
<html>
Follow the instructions in JnlpFileChooser.java to
build jnlpfilechooser.jar, then:
<a href="filechooser.jnlp">click here</a>
</html>
//:~
```

Una volta che avrete scaricato l'applicazione, potete configurarla tramite la console amministrativa. Se state utilizzando Java web Start in ambiente Windows, la seconda volta che eseguite l'applicazione il sistema vi chiederà se volete creare un collegamento: questo comportamento è comunque configurabile. In questo paragrafo sono stati esaminati solo due servizi offerti da JNLP, ma la versione corrente ne comprende sette; ciascuno di essi è progettato per eseguire operazioni specifiche, quali la stampa o la gestione degli appunti. Per ulteriori informazioni, consultate il sito <http://java.sun.com>.

## Concorrenza e Swing

Nella programmazione con le librerie Swing si utilizzano i thread. Questo argomento è stato accennato all'inizio del capitolo, quando si è detto che ogni operazione deve essere inviata al thread di invio degli eventi di Swing tramite il metodo **SwingUtilities.invokeLater()**. Tuttavia, il fatto di non dovere creare esplicitamente un oggetto **Thread** significa che eventuali problemi causati dai thread potrebbero trovarvi impreparati. Dovete avere ben presente che esiste un thread di invio degli eventi di Swing, sempre attivo, il quale gestisce tutti gli eventi Swing estraendoli uno per uno dalla coda degli eventi e mandandoli in esecuzione. Ricordandovi dell'esistenza di questo thread, sarete certi che la vostra applicazione non sarà soggetta a situazioni di stallo o di corsa critica.

I prossimi paragrafi tratteranno i problemi di threading che potrebbero presentarsi nell'utilizzo di Swing.



## ***Task di lunga durata***

Uno degli errori peggiori che potreste commettere programmando con un'interfaccia grafica è utilizzare inavvertitamente il thread di invio degli eventi per eseguire un task di lunga durata. Considerate questo semplice esempio:

```
//: gui/LongRunningTask.java
// Un programma mal progettato.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;

public class LongRunningTask extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task");
    public LongRunningTask() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                try {
                    TimeUnit.SECONDS.sleep(3);
                } catch(InterruptedException e) {
                    System.out.println("Task interrupted");
                    return;
                }
                System.out.println("Task completed");
            }
        });
        b2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                // Lo interrompevo voi?
                Thread.currentThread().interrupt();
            }
        });
        setLayout(new FlowLayout());
        add(b1);
```



```
        add(b2);
    }
    public static void main(String[] args) {
        run(new LongRunningTask(), 200, 150);
    }
} ///:~
```

Quando premete il pulsante **b1**, il thread di invio degli eventi si trova improvvisamente occupato nell'esecuzione di un task di lunga durata. Noterete che non è neppure possibile fare clic sul pulsante, perché il thread che normalmente ridisegnerebbe lo schermo è occupato. Non potete fare altro, neppure premere il pulsante **b2**, perché il programma non risponderà finché il task di **b1** non sarà completo e il thread di invio degli eventi nuovamente disponibile. Il codice presente in **b2** è soltanto un goffo tentativo di risolvere il problema interrompendo il thread di invio degli eventi.

La soluzione, naturalmente, consiste nell'eseguire i processi di lunga durata in thread separati. L'esempio seguente utilizza l'esecutore di thread singoli **Executor**, che accoda automaticamente i task in attesa e li esegue uno per volta:

```
//: gui/InterruptableLongRunningTask.java
// Task di lunga durata in thread separati.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;

class Task implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    public void run() {
        System.out.println(this + " started");
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch(InterruptedException e) {
            System.out.println(this + " interrupted");
            return;
        }
    }
}
```



```
        System.out.println(this + " completed");
    }
    public String toString() { return "Task " + id; }
    public long id() { return id; }
};

public class InterruptableLongRunningTask extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task");
    ExecutorService executor =
        Executors.newSingleThreadExecutor();
    public InterruptableLongRunningTask() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Task task = new Task();
                executor.execute(task);
                System.out.println(task + " added to the queue");
            }
        });
        b2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                executor.shutdownNow(); // Forzatura eccessiva
            }
        });
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
    }
    public static void main(String[] args) {
        run(new InterruptableLongRunningTask(), 200, 150);
    }
} //:-~
```

Questa tecnica è migliore, ma quando premete il pulsante **b2** viene chiamato **shutdownNow()** su **ExecutorService**, disabilitandolo; se provate ad aggiungere altri task, otterrete un'eccezione. Pertanto, la pressione del pulsante **b2**



rende il programma non operativo. È necessario invece interrompere il task corrente e annullare i task in attesa, senza arrestare tutto. Il meccanismo **Callable/Future** di Java SE5 descritto nel Capitolo 1 è proprio ciò che vi occorre. Definite una nuova classe chiamata **TaskManager**, contenente alcune *tuple* con l'oggetto **Callable** che rappresenta il task e l'oggetto **Future** restituito da **Callable**. Il motivo per cui sono necessarie le tuple è che permettono di tenere traccia del task originale, fornendo informazioni supplementari che non sono disponibili tramite **Future**. Osservate l'esempio seguente:

```
//: net/mindview/util/TaskItem.java
// Un Future e il Callable che lo produce.
package net.mindview.util;
import java.util.concurrent.*;

public class TaskItem<R,C extends Callable<R>> {
    public final Future<R> future;
    public final C task;
    public TaskItem(Future<R> future, C task) {
        this.future = future;
        this.task = task;
    }
} //:-
```

Nella libreria di **java.util.concurrent** il task non è disponibile tramite il **Future** in modo predefinito, perché non è detto che lo sia quando si ottengono i risultati da **Future**. In questo caso, il task viene memorizzato per essere sempre a disposizione.

**TaskManager** viene posto in **net.mindview.util**, così da essere disponibile come classe di utilità generale:

```
//: net/mindview/util/TaskManager.java
// Gestisce ed esegue una coda di task.
package net.mindview.util;
import java.util.concurrent.*;
import java.util.*;

public class TaskManager<R,C extends Callable<R>>
    extends ArrayList<TaskItem<R,C>> {
```



```
private ExecutorService exec =
    Executors.newSingleThreadExecutor();
public void add(C task) {
    add(new TaskItem<R,C>(exec.submit(task),task));
}
public List<R> getResults() {
    Iterator<TaskItem<R,C>> items = iterator();
    List<R> results = new ArrayList<R>();
    while(items.hasNext()) {
        TaskItem<R,C> item = items.next();
        if(item.future.isDone()) {
            try {
                results.add(item.future.get());
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
            items.remove();
        }
    }
    return results;
}
public List<String> purge() {
    Iterator<TaskItem<R,C>> items = iterator();
    List<String> results = new ArrayList<String>();
    while(items.hasNext()) {
        TaskItem<R,C> item = items.next();
        // Mantiene i task completati per restituire i risultati:
        if(!item.future.isDone()) {
            results.add("Cancelling " + item.task);
            item.future.cancel(true); // Puo' interrompere
            items.remove();
        }
    }
    return results;
}
} ///:~
```



**TaskManager** è un **ArrayList** di **TaskItem** che contiene anche un **Executor** di thread singoli, in modo che quando chiamate **add()** passando un oggetto **Callable**, invia il **Callable** e memorizza il **Future** risultante con il task originale. In questo modo, se dovete eseguire qualcosa con il task disporrete di un riferimento: come esempio, in **purge()** viene utilizzato il metodo **toString()** del task.

Questo meccanismo può essere utilizzato per gestire i task di lunga durata dell'esempio precedente:

```
//: gui/InterruptableLongRunningCallable.java
// Utilizzo di oggetti Callable per task di lunga durata.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class CallableTask extends Task
implements Callable<String> {
    public String call() {
        run();
        return "Return value of " + this;
    }
}

public class
InterruptableLongRunningCallable extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task"),
        b3 = new JButton("Get results");
    private TaskManager<String,CallableTask> manager =
        new TaskManager<String,CallableTask>();
    public InterruptableLongRunningCallable() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
```



```
CallableTask task = new CallableTask();
manager.add(task);
System.out.println(task + " added to the queue");
}
});
b2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        for(String result : manager.purge())
            System.out.println(result);
    }
});
b3.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Chiamata d'esempio a un metodo del task:
        for(TaskItem<String,CallableTask> tt :
            manager)
            tt.task.id(); // Nessun cast necessario
        for(String result : manager.getResults())
            System.out.println(result);
    }
});
setLayout(new FlowLayout());
add(b1);
add(b2);
add(b3);
}
}
public static void main(String[] args) {
    run(new InterruptableLongRunningCallable(), 200, 150);
}
}
} //:~
```

Come potete vedere, **CallableTask** esegue esattamente lo stesso processo di **Task** salvo poi restituire un risultato, in questo caso uno **String** che identifica il task.

I programmi di utilità non Swing chiamati *SwingWorker* (<http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html#SwingWorker>) e *Foxtrot* (<http://foxtrot.sourceforge.net>), assenti nella distribuzione Java standard,



sono stati creati per risolvere un problema simile; tuttavia, al momento della stesura di questo manuale non sono stati aggiornati per trarre vantaggio dal meccanismo **Callable/Future** di Java SE5.

Spesso è importante che l'utente finale abbia un'indicazione visiva dell'esecuzione di un task e del suo progresso. Normalmente, si utilizza un componente **JProgressBar** o **ProgressMonitor**; l'esempio seguente si serve di **ProgressMonitor**.

```
//: gui/MonitoredLongRunningCallable.java
// Visualizza il progresso di un task con ProgressMonitor.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class MonitoredCallable implements Callable<String> {
    private static int counter = 0;
    private final int id = counter++;
    private final ProgressMonitor monitor;
    private final static int MAX = 8;
    public MonitoredCallable(ProgressMonitor monitor) {
        this.monitor = monitor;
        monitor.setNote(toString());
        monitor.setMaximum(MAX - 1);
        monitor.setMillisToPopup(500);
    }
    public String call() {
        System.out.println(this + " started");
        try {
            for(int i = 0; i < MAX; i++) {
                TimeUnit.MILLISECONDS.sleep(500);
                if(monitor.isCanceled())
                    Thread.currentThread().interrupt();
                final int progress = i;
                SwingUtilities.invokeLater(
                    new Runnable() {

```



```
    public void run() {
        monitor.setProgress(progress);
    }
}
);
}
} catch(InterruptedException e) {
    monitor.close();
    System.out.println(this + " interrupted");
    return "Result: " + this + " interrupted";
}
System.out.println(this + " completed");
return "Result: " + this + " completed";
}
public String toString() { return "Task " + id; }
};

public class MonitoredLongRunningCallable extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task"),
        b3 = new JButton("Get results");
    private TaskManager<String,MonitoredCallable> manager =
        new TaskManager<String,MonitoredCallable>();
    public MonitoredLongRunningCallable() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                MonitoredCallable task = new MonitoredCallable(
                    new ProgressMonitor(
                        MonitoredLongRunningCallable.this,
                        "Long-Running Task", "", 0, 0)
                );
                manager.add(task);
                System.out.println(task + " added to the queue");
            }
        });
    }
}
```



```
b2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        for(String result : manager.purge())
            System.out.println(result);
    }
});
b3.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        for(String result : manager.getResults())
            System.out.println(result);
    }
});
setLayout(new FlowLayout());
add(b1);
add(b2);
add(b3);
}
public static void main(String[] args) {
    run(new MonitoredLongRunningCallable(), 200, 500);
}
} //:-~
```

Il costruttore **MonitoredCallable** accetta come argomento un **ProgressMonitor**, e il relativo metodo **call()** aggiorna il **ProgressMonitor** ogni mezzo secondo. Notate che **MonitoredCallable** è un task separato e non dovrebbe tentare di gestire direttamente l'interfaccia grafica; viene pertanto utilizzato **SwingUtilities.invokeLater()** per inviare le informazioni sullo stato di avanzamento al **monitor**. Il Tutorial di Swing fornito da Sun (<http://java.sun.com/docs/books/tutorial/uiswing/>) presenta un approccio alternativo servendosi di un **Timer** Swing, che controlla la condizione del task e aggiorna il monitor.

Se viene premuto il pulsante “annulla”, il metodo **monitor.isCanceled()** restituirà **true**. In questo caso, il task chiama semplicemente **interrupt()** sul proprio thread che terminerà nella clausola **catch**, in cui il **monitor** viene chiuso con il metodo **close()**. Il resto del codice è lo stesso dell'esempio precedente, tranne nella creazione del **ProgressMonitor** come parte del costruttore di **MonitoredLongRunningCallable**.

**Esercizio 33 (6)** Modificate **InterruptibleLongRunningCallable.java** in modo che esegua tutti i task parallelmente, invece che in sequenza.



## Threading visuale

Il seguente esempio crea una classe **Runnable JPanel** che assegna alla propria finestra colori differenti. Questa applicazione è impostata per accettare dalla riga di comando valori che determinano le dimensioni della griglia di colori e la durata della pausa (**sleep()**) tra i cambi di colore. Cambiando questi valori scoprirete alcune caratteristiche interessanti, forse inspiegabili, nell'implementazione del threading sulla vostra piattaforma:

```
//: gui/ColorBoxes.java
// Una dimostrazione visuale del threading.
// {Args: 12 50}
import javax.swing.*;
import java.awt.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

class CBox extends JPanel implements Runnable {
    private int pause;
    private static Random rand = new Random();
    private Color color = new Color(0);
    public void paintComponent(Graphics g) {
        g.setColor(color);
        Dimension s = getSize();
        g.fillRect(0, 0, s.width, s.height);
    }
    public CBox(int pause) { this.pause = pause; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                color = new Color(rand.nextInt(0xFFFFFF));
                repaint(); // Richiesta asincrona a paint()
                TimeUnit.MILLISECONDS.sleep(pause);
            }
        } catch(InterruptedException e) {
            // Un modo accettabile per uscire
        }
    }
}
```



```
    }
}

public class ColorBoxes extends JFrame {
    private int grid = 12;
    private int pause = 50;
    private static ExecutorService exec =
        Executors.newCachedThreadPool();
    public ColorBoxes() {
        setLayout(new GridLayout(grid, grid));
        for(int i = 0; i < grid * grid; i++) {
            CBox cb = new CBox(pause);
            add(cb);
            exec.execute(cb);
        }
    }
    public static void main(String[] args) {
        ColorBoxes boxes = new ColorBoxes();
        if(args.length > 0)
            boxes.grid = new Integer(args[0]);
        if(args.length > 1)
            boxes.pause = new Integer(args[1]);
        run(boxes, 500, 400);
    }
} //:-~
```

**ColorBoxes** configura un **GridLayout** in modo che sia composto da una griglia di celle **grid**; poi aggiunge il numero adatto di oggetti **CBox** per riempire questa griglia, passando il valore **pause** a ogni cella. In **main()** potete notare che **pause** e **grid** possiedono valori predefiniti, che possono essere modificati fornendo gli argomenti opportuni da riga di comando.

In **CBox** è gestito tutto il carico operativo. Questa classe è ereditata da **JPanel** e implementa l'interfaccia **Runnable**, per fare in modo che anche **JPanel** sia un task indipendente. Questi task sono pilotati da un thread pool, **ExecutorService**.

Il colore corrente delle celle è **color**; i colori vengono creati mediante il costruttore **Color**, che accetta valori a 24 bit, in questo caso generati casualmente.



Il metodo **paintComponent()** imposta semplicemente il colore a **color**, poi riempie l'intero **JPanel** con questo colore.

In **run()** potete vedere il ciclo infinito che imposta la variabile **color** a un nuovo colore generato casualmente, e la chiamata a **repaint()** per visualizzarlo. A quel punto, il thread si pone in attesa (**sleep()**) per il tempo indicato dalla riga di comando.

La chiamata di **repaint()** in **run()** merita di essere analizzata. A una prima occhiata può sembrare che vengano creati numerosi thread, ognuno dei quali forza una colorazione. Potreste ritenere che questo violi il principio che impone di sottoporre i thread alla coda di eventi, ma in realtà questi thread non stanno modificando la risorsa condivisa. Quando chiamano **repaint()**, non forzano la colorazione in quel preciso istante, ma si limitano a impostare un flag per indicare che la prossima volta che il thread di invio degli eventi sarà pronto a colorare i riquadri, l'area corrente sarà candidata per la ricolorazione: in questo modo il programma non causerà problemi di threading con Swing.

Quando il thread di invio degli eventi esegue una chiamata a **paint()**, in primo luogo chiama **paintComponent()**, poi **paintBorder()** e **paintChildren()**. Per sovrascrivere **paint()** in un componente derivato, dovrete ricordarvi di chiamare la versione della classe di base di **paint()**, per fare in modo che vengano eseguite le operazioni opportune.

Proprio perché questo design è flessibile e il threading è collegato a ogni elemento **JPanel**, avete la possibilità di sperimentare creando tutti i thread che desiderate. Tenete presente che, in realtà, esiste un limite al numero di thread che possono essere gestiti dalla vostra JVM.

Questo programma è anche un interessante strumento di test, perché può evidenziare le notevoli differenze di prestazioni e di comportamento nell'implementazione del threading tra le diverse piattaforme e versioni di JVM.

**Esercizio 34 (4)** Modificate **ColorBoxes.java** in modo che inizi generando dei punti diffusi ("stelle") sulla finestra; il programma dovrà poi modificare casualmente i colori di queste "stelle".

## Programmazione visuale e JavaBeans

Finora, in questo manuale avete visto l'importanza di Java nella creazione di porzioni di codice riutilizzabili. L'unità di codice "più riutilizzabile" è sempre stata la classe, in quanto comprende un'unità coesiva di caratteristiche (campi) e comportamenti (metodi) che possono essere riutilizzati direttamente, tramite la composizione o per ereditarietà.

L'ereditarietà e il polimorfismo sono elementi essenziali della programmazione orientata agli oggetti ma, nella maggior parte dei casi, quando state costruendo un'applicazione ciò che realmente desiderate è avere *componenti* che facciano esattamente quanto vi occorre. Vorreste poter collegare queste parti nel vostro progetto come un tecnico elettronico monta i vari componenti su un circuito stampato. Deve necessariamente esistere un metodo che acceleri questo stile di programmazione "modulare".

La "programmazione visuale" iniziò a diventare molto diffusa con Microsoft Visual BASIC (VB), seguito da un progetto di seconda generazione, Delphi di Borland. Quest'ultimo è stato la fonte d'ispirazione principale per la progettazione della struttura JavaBeans. In questi strumenti di programmazione i componenti vengono rappresentati visivamente, il che ha perfettamente senso dal momento che si riferiscono a componenti visivi, quali un pulsante o un campo di testo. La rappresentazione visiva, infatti, spesso corrisponde esattamente al componente che sarà visualizzato durante l'esecuzione del programma. Parte del processo di programmazione visuale richiede di trascinare e incollare un componente da una tavolozza, per disporlo sul form. Gli strumenti IDE (*Integrated Development Environment*) scrivono il codice necessario e questo codice fa in modo che il componente venga creato quando si esegue il programma.

Disporre il componente su un form non è certamente sufficiente per completare il programma: spesso, infatti, dovete modificare le caratteristiche di un componente, per esempio il colore, il testo, il database cui è collegato un elemento ecc. Le caratteristiche che possono essere modificate in fase di progettazione sono dette *proprietà*. Potete manipolare le proprietà del vostro componente all'interno dell'IDE; quando create il programma, questi dati di configurazione vengono salvati in modo da essere ripristinati in fase di esecuzione.

Ormai siete abituati all'idea che un oggetto è più dell'insieme delle proprie caratteristiche: è anche un insieme di comportamenti. Durante la progettazione i comportamenti di un componente visuale sono parzialmente rappresentati dagli *eventi*, che indicano "ciò che può accadere" al componente. Di norma, decidete voi che cosa volette accada al verificarsi di un certo evento, collegandogli codice appropriato.

Questa è la fase critica: l'IDE utilizza la riflessione per consultare dinamicamente il componente e scoprire le proprietà e gli eventi che lo caratterizzano. Una volta che l'IDE conosce queste informazioni, può non soltanto visualizzare le proprietà e permettervi di modificarle (salvandone lo stato quando compilate il programma), ma anche mostrarvi gli eventi. Generalmente con questi software IDE è sufficiente eseguire un'operazione banale, per esempio



il doppio clic sul nome di un evento, per creare automaticamente il corpo del codice già collegato a quel particolare evento: a quel punto, non dovete fare altro che scrivere il codice da eseguire quando l'evento si manifesta.

Tutto questo comporta molto lavoro aggiuntivo, che viene eseguito per voi dall'IDE. Di conseguenza, potete focalizzare la vostra attenzione sull'aspetto applicativo e sulle funzionalità, sapendo che l'IDE s'incaricherà di gestire i dettagli implementativi. Il motivo del grande successo degli strumenti di programmazione visuale è che accelerano enormemente il processo di sviluppo delle applicazioni: sicuramente quello dell'interfaccia grafica, ma spesso anche di altri componenti applicativi.

### Che cos'è JavaBean?

In fondo, un componente non è che un blocco di codice, solitamente incorporato in una classe. Il punto chiave è la capacità dell'IDE di scoprire le proprietà e gli eventi di quel componente. Per creare un componente VB, in origine il programmatore doveva scrivere complesse porzioni di codice seguendo le convenzioni opportune per esporre le proprietà e gli eventi: tale procedura si è poi semplificata nel corso del tempo. Delphi è uno strumento di programmazione visuale di seconda generazione: il linguaggio è stato concepito appositamente per questo tipo di programmazione, rendendo molto più facile la creazione di un componente visuale. Tuttavia Java, con la tecnologia JavaBean, ha portato la creazione dei componenti visuali a uno stadio più avanzato, perché un bean è una classe. Non dovete scrivere alcun codice aggiuntivo né utilizzare estensioni particolari del linguaggio per creare un bean. Tutto ciò che dovete fare, infatti, è una piccola modifica alla nomenclatura dei vostri metodi. È il nome del metodo che indica all'IDE se qualcosa è una proprietà, un evento, o un normale metodo.

Nella documentazione JDK, questa convenzione di nomenclatura viene erroneamente chiamata "design pattern". Questo è un termine inopportuno, perché i design pattern, che potete trovare in *Thinking in Patterns (with Java)* sul sito [www.mindview.net](http://www.mindview.net), sono già abbastanza impegnativi senza questa confusione. Non si tratta di un design pattern, ma di una convenzione di nomenclatura, anche piuttosto semplice.

1. Per una proprietà chiamata **xxx**, dovete creare due metodi: **getXXX()** e **setXXX()**. La prima lettera dopo "get" o "set" viene automaticamente convertita in minuscolo da tutti gli strumenti che estraggono i metodi, generando così il nome della proprietà. Il tipo prodotto dal metodo "get" è lo stesso del tipo dell'argomento passato al metodo "set". Il nome della proprietà e il tipo per i metodi "get" e "set" non sono in relazione.



2. Per una proprietà di tipo **boolean**, oltre ai metodi “get” e “set” descritti potete anche utilizzare “is” in luogo di “get”.
3. I normali metodi del bean non si conformano a questa convenzione di nomenclatura, ma devono essere **public**.
4. Per gli eventi, adottate l’approccio “listener” di Swing. È esattamente lo stesso che avete osservato: **addBounceListener(BounceListener)** e **removeBounceListener (BounceListener)** per gestire un **BounceEvent**. Di solito gli eventi e i listener incorporati soddisferanno le vostre necessità, ma potrete avere la necessità di creare i vostri eventi e interfacce listener.

Servendovi di queste informazioni come riferimento, potete creare un semplice bean:

```
//: frogbean/Frog.java
// Un banale JavaBean.
package frogbean;
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
    private int jumps;
    private Color color;
    private Spots spots;
    private boolean jmpr;
    public int getJumps() { return jumps; }
    public void setJumps(int newJumps) {
        jumps = newJumps;
    }
    public Color getColor() { return color; }
    public void setColor(Color newColor) {
        color = newColor;
    }
    public Spots getSpots() { return spots; }
    public void setSpots(Spots newSpots) {
        spots = newSpots;
    }
}
```



```
public boolean isJumper() { return jmpr; }
public void setJumper(boolean j) { jmpr = j; }
public void addActionListener(ActionListener l) {
    //...
}
public void removeActionListener(ActionListener l) {
    // ...
}
public void addKeyListener(KeyListener l) {
    // ...
}
public void removeKeyListener(KeyListener l) {
    // ...
}
// Un "normale" metodo public:
public void croak() {
    System.out.println("Ribbet!");
}
} //:~
```

In primo luogo, potete osservare che si tratta di una classe. Normalmente, tutti i vostri campi saranno **private**, accessibili soltanto attraverso i metodi e le proprietà. Secondo la convenzione di nomenclatura, le proprietà sono **jumps**, **color**, **points** e **jumper**: notate il cambiamento della prima lettera nel nome della proprietà, da maiuscola a minuscola. Nonostante il nome dell'identificativo interno equivalga al nome della proprietà nei primi tre casi, in **jumper** potete notare che il nome della proprietà non forza l'utilizzo di alcun identificativo particolare per le variabili interne, né obbliga in realtà ad *avere* una variabile interna per quella proprietà.

Gli eventi gestiti da questo bean sono **ActionEvent** e **KeyEvent**, basati sulla nomenclatura dei metodi "add" e "remove" per il listener collegato. Infine, potete vedere che il normale metodo **croak()** è anch'esso parte del bean perché è un metodo **public**, non perché sia conforme allo schema di nomenclatura.

### **Estrazione di BeanInfo con Introspector**

Una delle parti critiche dello schema di JavaBean si evidenzia trascinando un bean dalla tavolozza e inserendolo in un form. L'IDE deve essere in grado di creare il bean (operazione fattibile se è presente un costruttore



re predefinito) e quindi, senza accedere al codice sorgente del bean, deve estrarre tutte le informazioni necessarie per creare l'elenco delle proprietà e i gestori degli eventi.

Parte della soluzione è già evidente da quanto trattato nel Volume 2, Capitolo 2: la *riflessione* di Java identifica tutti i metodi in una classe sconosciuta. Questa tecnica risolve perfettamente il problema dei JavaBean senza richiedere l'implementazione di parole chiave supplementari nel linguaggio, come accade per altri linguaggi di programmazione visuali. Infatti, una delle ragioni principali per cui la riflessione è stata aggiunta al linguaggio Java era il supporto alla tecnologia JavaBean. In ogni caso, la riflessione supporta anche la serializzazione degli oggetti e la tecnologia RMI (*Remote Method Invocation*), ed è utile anche nella normale programmazione. Potete quindi supporre che l'IDE utilizzi la riflessione su ogni bean per cercare tra i suoi metodi le proprietà e gli eventi del bean in questione.

Questo è possibile, ma i progettisti di Java desideravano fornire anche uno strumento standard, non soltanto per rendere i bean più semplici da utilizzare, ma anche per fornire uno standard utile per la creazione di bean più complessi. Questo strumento è rappresentato dalla classe **Introspector**, il cui metodo principale è **static getBeanInfo()**. Quando passate un riferimento **Class** a questo metodo, esso consulta la classe e restituisce un oggetto **BeanInfo** in cui potete trovare le proprietà, i metodi e gli eventi della classe.

In generale, non dovrete preoccuparvi di eseguire questa operazione: con ogni probabilità la maggior parte dei vostri bean sarà immediatamente disponibile, quindi non avrete necessità di conoscere in dettaglio tutti i meccanismi necessari al loro funzionamento. Vi basterà trascinare i bean sul vostro form, configurarne le proprietà e scrivere i gestori per gli eventi che vi interessano. In ogni caso, l'utilizzo di Introspector per visualizzare le informazioni di un bean è certamente un'esperienza educativa. Ecco un programma che esegue questa operazione:

```
//: gui/BeanDumper.java
// Introspezione di un bean.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingConsole.*;
```



```
public class BeanDumper extends JFrame {
    private JTextField query = new JTextField(20);
    private JTextArea results = new JTextArea();
    public void print(String s) { results.append(s + "\n"); }
    public void dump(Class<?> bean) {
        results.setText("");
        BeanInfo bi = null;
        try {
            bi = Introspector.getBeanInfo(bean, Object.class);
        } catch(IntrospectionException e) {
            print("Couldn't introspect " + bean.getName());
            return;
        }
        for(PropertyDescriptor d: bi.getPropertyDescriptors()){
            Class<?> p = d.getPropertyType();
            if(p == null) continue;
            print("Property type:\n " + p.getName() +
                  "Property name:\n " + d.getName());
            Method readMethod = d.getReadMethod();
            if(readMethod != null)
                print("Read method:\n " + readMethod);
            Method writeMethod = d.getWriteMethod();
            if(writeMethod != null)
                print("Write method:\n " + writeMethod);
            print("=====");
        }
        print("Public methods:");
        for(MethodDescriptor m : bi.getMethodDescriptors())
            print(m.getMethod().toString());
        print("=====");
        print("Event support:");
        for(EventSetDescriptor e: bi.getEventSetDescriptors()){
            print("Listener type:\n " +
                  e.getListenerType().getName());
            for(Method lm : e.getListenerMethods())
                print("Listener method:\n " + lm.getName());
        }
    }
}
```



```
for(MethodDescriptor lmd :  
    e.getListenerMethodDescriptors() )  
    print("Method descriptor:\n " + lmd.getMethod());  
Method addListener= e.getAddListenerMethod();  
print("Add Listener Method:\n " + addlistener);  
Method removeListener = e.getRemoveListenerMethod();  
print("Remove Listener Method:\n " + removeListener);  
print("=====");  
}  
}  
class Dumper implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
        String name = query.getText();  
        Class<?> c = null;  
        try {  
            c = Class.forName(name);  
        } catch(ClassNotFoundException ex) {  
            results.setText("Couldn't find " + name);  
            return;  
        }  
        dump(c);  
    }  
}  
public BeanDumper() {  
    JPanel p = new JPanel();  
    p.setLayout(new FlowLayout());  
    p.add(new JLabel("Qualified bean name:"));  
    p.add(query);  
    add(BorderLayout.NORTH, p);  
    add(new JScrollPane(results));  
    Dumper dmpr = new Dumper();  
    query.addActionListener(dmpr);  
    query.setText("frogbean.Frog");  
    // Forza la valutazione  
    dmpr.actionPerformed(new ActionEvent(dmpr, 0, ""));  
}
```



```
public static void main(String[] args) {
    run(new BeanDumper(), 600, 500);
}
} //:~
```

Il metodo **BeanDumper.dump()** esegue tutto il lavoro. In primo luogo, cerca di creare un oggetto **BeanInfo**: se l'operazione ha successo, chiama i metodi di **BeanInfo** che producono le informazioni relative alle proprietà, ai metodi e agli eventi. Come vedete, nella chiamata a **Introspector.getBeanInfo()** è presente un secondo argomento che indica a **Introspector** dove arrestarsi nella gerarchia ereditaria: in questo caso, si ferma prima di analizzare i metodi di **Object**, in quanto non sono considerati interessanti.

Per le proprietà, **getPropertyDescriptors()** restituisce un array di **PropertyDescriptor**, per ciascuno dei quali potete chiamare **getPropertyType()** al fine di trovare la classe dell'oggetto cui fanno riferimento, in lettura o scrittura, i metodi delle proprietà. In seguito, per ogni proprietà, potete ottenere il relativo pseudonimo (estratto dal nome del metodo) tramite **getName()**, il metodo per la lettura con **getReadMethod()** e quello per la scrittura con **getWriteMethod()**. Questi ultimi due metodi restituiscono un oggetto **Method** utilizzabile per invocare il metodo corrispondente sull'oggetto: questa tecnica fa parte della riflessione.

Per i metodi **public**, compresi i metodi delle proprietà, **getMethodDescriptors()** restituisce un array di **MethodDescriptor**, da ciascuno dei quali potete ottenere l'oggetto **Method** associato e visualizzarne il nome.

Per gli eventi, **getEventSetDescriptors()** restituisce un array di **EventSetDescriptor**, ciascuno dei quali può essere interrogato per ottenere la classe del listener, i metodi di questa classe listener e i metodi per l'aggiunta e la rimozione di listener. Il programma **BeanDumper** visualizza tutte queste informazioni.

All'avvio, il programma forza la valutazione di **frogbean.Frog**. Dopo la rimozione di alcuni dettagli insignificanti, l'output è il seguente:

```
Property type:
Color
Property name:
color
Read method:
public Color getColor()
Write method:
public void setColor(Color)
```



```
=====
Property type:
    boolean
Property name:
    jumper
Read method:
    public boolean isJumper()
Write method:
    public void setJumper(boolean)
=====

Property type:
    int
Property name:
    jumps
Read method:
    public int getJumps()
Write method:
    public void setJumps(int)
=====

Property type:
    frogbean.Spots
Property name:
    spots
Read method:
    public frogbean.Spots getSpots()
Write method:
    public void setSpots(frogbean.Spots)
=====

Public methods:
public void setSpots(frogbean.Spots)
public void setColor(Color)
public void setJumps(int)
public boolean isJumper()
public frogbean.Spots getSpots()
public void croak()
public void addActionListener(ActionListener)
public void addKeyListener(KeyListener)
```



```
public Color getColor()
public void setJumper(boolean)
public int getJumps()
public void removeActionListener(ActionListener)
public void removeKeyListener(KeyListener)

=====
Event support:
Listener type:
    Keylistener
Listener method:
    keyPressed
Listener method:
    keyReleased
Listener method:
    keyTyped
Method descriptor:
    public abstract void keyPressed(KeyEvent)
Method descriptor:
    public abstract void keyReleased(KeyEvent)
Method descriptor:
    public abstract void keyTyped(KeyEvent)
Add Listener Method:
    public void addKeyListener(KeyListener)
Remove Listener Method:
    public void removeKeyListener(KeyListener)

=====
Listener type:
    ActionListener
Listener method:
    actionPerformed
Method descriptor:
    public abstract void actionPerformed(ActionEvent)
Add Listener Method:
    public void addActionListener(ActionListener)
Remove Listener Method:
    public void removeActionListener(ActionListener)
```



Questo output rivela gran parte di ciò che **Introspector** scopre mentre produce un oggetto **BeanInfo** dal vostro bean. Potete vedere che il tipo di proprietà e il nome sono indipendenti; notate il nome della proprietà in minuscolo: possono essere presenti lettere maiuscole soltanto se il nome della proprietà contiene più di una maiuscola. Ricordate che i nomi di metodo visualizzati in questo output, così come quelli dei metodi di lettura e scrittura, sono prodotti da un oggetto **Method** che può essere utilizzato per invocare il metodo associato sull'oggetto.

L'elenco dei metodi **public** include sia quelli che non sono associati a una proprietà o a un evento, come il metodo **croak()**, sia quelli che lo sono. Questi sono tutti i metodi richiamabili per un bean; alcuni IDE li elencano mentre state creando le chiamate di metodo, per facilitare il vostro lavoro.

In conclusione, potete notare che gli eventi vengono analizzati suddividendoli in listener, metodi listener e metodi di aggiunta e rimozione listener. In pratica, una volta che disponete dell'oggetto **BeanInfo** potete scoprire tutte le caratteristiche importanti del bean. Potete anche chiamarne i metodi, anche se non avete altre informazioni oltre all'oggetto stesso: anche in questo caso, si tratta di una funzione offerta dalla riflessione.

### ***Un bean più complesso***

Questo esempio è leggermente più complesso, benché più frivolo. Si tratta di un **JPanel** che disegna un piccolo cerchio intorno al puntatore del mouse, quando questo viene spostato: quando fate clic con il mouse, la parola "Bang!" compare al centro della finestra e viene attivato un action listener.

Le proprietà modificabili sono rappresentate dalla dimensione del cerchio e da colore, dimensione e testo della parola che viene visualizzata al clic del mouse. Dal momento che **BangBean** dispone dei propri metodi **addActionListener()** e **removeActionListener()**, potete collegare il vostro listener, che verrà avviato quando l'utente farà clic sul componente **BangBean**. Dovreste riconoscere il supporto alle proprietà e agli eventi:

```
//: bangbean/BangBean.java
// Un bean grafico.
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
```



```
public class
BangBean extends JPanel implements Serializable {
    private int xm, ym;
    private int cSize = 20; // Dimensione del cerchio
    private String text = "Bang!";
    private int fontSize = 48;
    private Color tColor = Color.RED;
    private ActionListener actionListener;
    public BangBean() {
        addMouseListener(new ML());
        addMouseMotionListener(new MML());
    }
    public int getCircleSize() { return cSize; }
    public void setCircleSize(int newSize) {
        cSize = newSize;
    }
    public String getBangText() { return text; }
    public void setBangText(String newText) {
        text = newText;
    }
    public int getFontSize() { return fontSize; }
    public void setFontSize(int newSize) {
        fontSize = newSize;
    }
    public Color getTextColor() { return tColor; }
    public void setTextColor(Color newColor) {
        tColor = newColor;
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.BLACK);
        g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
    }
    // Questo e' un Listener unicast, la forma
    // piu' semplice di gestione di un Listener:
    public void addActionListener(ActionListener l)
```



```
throws TooManyListenersException {
    if(actionListener != null)
        throw new TooManyListenersException();
    actionListener = 1;
}
public void removeActionListener(ActionListener l) {
    actionListener = null;
}
class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(tColor);
        g.setFont(
            new Font("TimesRoman", Font.BOLD, fontSize));
        int width = g.getFontMetrics().stringWidth(text);
        g.drawString(text, (getSize().width - width) / 2,
                    getSize().height/2);
        g.dispose();
        // Chiama il metodo del listener:
        if(actionListener != null)
            actionListener.actionPerformed(
                new ActionEvent(BangBean.this,
                               ActionEvent.ACTION_PERFORMED, null));
    }
}
class MML extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}
public Dimension getPreferredSize() {
    return new Dimension(200, 200);
}
} ///:-
```



La prima cosa da notare è che **BangBean** implementa l'interfaccia **Serializable**. Questo significa che l'IDE utilizzerà la serializzazione per "conservare" tutte le informazioni relative a **BangBean**, dopo che l'utente del vostro bean avrà registrato i valori delle proprietà. Quando create il bean come componente dell'applicazione corrente, le proprietà "conservate" vengono ripristinate, permettendovi di ottenere esattamente ciò che avevate progettato.

Osservando la segnatura di **addActionListener()**, vedrete che il metodo può sollevare un'eccezione **TooManyListenersException**. Questo indica che il metodo **addActionListener()** è di tipo *unicast*, vale a dire che dell'evento verificatosi viene informato soltanto un listener. Di norma farete invece utilizzo di eventi *multicast*, in modo che più listener possano essere informati di un evento; il multicast presenta tuttavia alcuni problemi di threading, che verranno esaminati nel prossimo paragrafo, "JavaBeans e sincronizzazione". Nel frattempo, il ricorso all'unicast vi permetterà di aggirare il problema.

Quando fate clic con il mouse, al centro di **BangBean** viene visualizzato del testo; se il campo **actionListener** non è **null** viene chiamato **actionPerformed()**, che crea un nuovo oggetto **ActionEvent** nel processo. A ogni spostamento del mouse sono rilevate le nuove coordinate e la finestra viene ridisegnata, cancellando anche il testo eventualmente presente. Servendovi della classe **BangBeanTest** potete testare il bean:

```
//: bangbean/BangBeanTest.java
// {Timeout: 5} Interrompe il test dopo 5 secondi
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBeanTest extends JFrame {
    private JTextField txt = new JTextField(20);
    // Durante il test, mostra le azioni:
    class BBL implements ActionListener {
        private int count = 0;
        public void actionPerformed(ActionEvent e) {
            txt.setText("BangBean action " + count++);
        }
    }
}
```



```

public BangBeanTest() {
    BangBean bb = new BangBean();
    try {
        bb.addActionListener(new BBL());
    } catch(TooManyListenersException e) {
        txt.setText("Too many listeners");
    }
    add(bb);
    add(BorderLayout.SOUTH, txt);
}
public static void main(String[] args) {
    run(new BangBeanTest(), 400, 500);
}
} //:-

```

Quando un bean viene utilizzato tramite un'IDE questa classe non sarà necessaria, ma è utile per fornirvi un meccanismo di collaudo rapido per ciascuno dei vostri bean. **BangBeanTest** inserisce un **BangBean** all'interno di un **JFrame**, collegando un semplice **ActionListener** a **BangBean** per visualizzare un conteggio di eventi in **JTextField** ogni volta che si verifica un **ActionEvent**. Di solito, naturalmente, l'IDE crea la maggior parte del codice necessario per utilizzare il bean.

Quando eseguite **BangBean** tramite **BeanDumper** o inserite il **BangBean** all'interno di un ambiente di sviluppo compatibile con i bean, noterete che esistono molte altre proprietà ed eventi che sono evidenziati dal codice precedente: la ragione è che **BangBean** è ereditato da **JPanel** e che anche **JPanel** è un bean, pertanto è possibile vederne i relativi eventi e proprietà.

**Esercizio 35** (6) Procuratevi uno o più ambienti IDE scegliendo tra quelli gratuiti disponibili su Internet, oppure utilizzate un ambiente IDE commerciale. Scoprite come aggiungere il bean **BangBean** a questo ambiente per utilizzarlo.

### *JavaBeans e sincronizzazione*

Quando create un bean, dovete presumere che lavori in un ambiente multi-thread. Questo aspetto ha determinati risvolti elencati di seguito.

1. Quando possibile, tutti i metodi **public** di un bean dovrebbero essere **synchronized**; ovviamente in fase di esecuzione **synchronized** comporta un onere aggiuntivo, seppure ridotto nelle versioni più recenti del JDK. Se



questo onore è eccessivo, i metodi che non causano problemi nelle sezioni critiche del codice non potranno essere definiti come **synchronized**; ricordate, tuttavia, che tali metodi non sono sempre facilmente riconoscibili. I metodi di questo tipo sono generalmente molto brevi, come **getCircleSize()** nell'esempio seguente, e/o "atomici": in pratica, la chiamata al metodo dovrebbe essere eseguita in una quantità di codice talmente ridotta così che l'oggetto non possa essere cambiato durante l'esecuzione. In ogni caso consultate il Capitolo 1, perché le operazioni che ritenete atomiche potrebbero non esserlo. Eseguire questi metodi senza la modalità **synchronized** potrebbe non avere un effetto decisivo sulla velocità di esecuzione del programma. È consigliabile che rendiate **synchronized** tutti i metodi **public** di un bean, e poi rimuoviate questa parola chiave dalla definizione di un metodo soltanto quando sarete certi che tale operazione produrrà effetti positivi e sarà sicura.

2. Quando attivate un evento multicast per un gruppo di listener interessati a quell'evento, dovete tenete presente che tali listener potrebbero essere aggiunti o rimossi mentre ne scorrete l'elenco.

Il primo punto è piuttosto semplice, mentre il secondo richiede un'attenta riflessione. **BangBean.java** ha evitato il problema della concorrenza ignorando la parola chiave **synchronized** e rendendo l'evento unicast. La versione seguente, invece, è stata modificata per lavorare in ambiente multithread, e si serve di eventi multicast:

```
//: gui/BangBean2.java
// Dovete scrivere i vostri bean in questo modo
// per eseguirli in ambiente multithreaded.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBean2 extends JPanel
implements Serializable {
    private int xm, ym;
    private int cSize = 20; // Dimensione del cerchio
    private String text = "Bang!";
    private int fontSize = 48;
```



```
private Color tColor = Color.RED;
private ArrayList<ActionListener> actionListeners =
    new ArrayList<ActionListener>();
public BangBean2() {
    addMouseListener(new ML());
    addMouseMotionListener(new MM());
}
public synchronized int getCircleSize() { return cSize; }
public synchronized void setCircleSize(int newSize) {
    cSize = newSize;
}
public synchronized String getBangText() { return text; }
public synchronized void setBangText(String newText) {
    text = newText;
}
public synchronized int getFontSize(){ return fontSize; }
public synchronized void setFontSize(int newSize) {
    fontSize = newSize;
}
public synchronized Color getTextColor(){ return tColor;}
public synchronized void setTextColor(Color newColor) {
    tColor = newColor;
}
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.BLACK);
    g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
}
// Questo e' un listener multicast, piu' frequente
// dell'approccio unicast di BangBean.java:
public synchronized void
addActionListener(ActionListener l) {
    actionListeners.add(l);
}
public synchronized void
removeActionListener(ActionListener l) {
    actionListeners.remove(l);
}
```



```
}

// Notate che non e' synchronized:
public void notifyListeners() {
    ActionEvent a = new ActionEvent(BangBean2.this,
        ActionEvent.ACTION_PERFORMED, null);
    ArrayList<ActionListener> lv = null;
    // Si crea una copia della List nel caso, mentre vengono
    // chiamati i listener, ne venga aggiunto uno:
    synchronized(this) {
        lv = new ArrayList<ActionListener>(actionListeners);
    }
    // Chiama tutti i metodi listener:
    for(ActionListener al : lv)
        al.actionPerformed(a);
}

class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(tColor);
        g.setFont(
            new Font("TimesRoman", Font.BOLD, fontSize));
        int width = g.getFontMetrics().stringWidth(text);
        g.drawString(text, (getSize().width - width) / 2,
            getSize().height/2);
        g.dispose();
        notifyListeners();
    }
}

class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}

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



```
BangBean2 bb2 = new BangBean2();
bb2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("ActionEvent" + e);
    }
});
bb2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("BangBean2 action");
    }
});
bb2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("More action");
    }
});
JFrame frame = new JFrame();
frame.add(bb2);
run(frame, 300, 300);
}
} //:-~
```

L'aggiunta della parola chiave **synchronized** ai metodi è una modifica semplice. Notate che nei metodi **addActionListener()** e **removeActionListener()** ora gli **ActionListener** vengono aggiunti e rimossi da un **ArrayList**, così da poter averne un numero a piacere.

Come potete vedere, il metodo **notifyListeners()** non è **synchronized**; esso può essere chiamato da più di un thread per volta. I metodi **addActionListener()** o **removeActionListener()** possono anche essere chiamati nel corso di una chiamata a **notifyListeners()**, il che è un problema poiché questo metodo sta scorrendo l'**ArrayList** **actionListeners**.

Per limitare il problema, l'**ArrayList** viene duplicato in un blocco di codice **synchronized** e ne viene letto il clone; per maggiori informazioni sulla clonazione, consultate i supplementi in linea di questo manuale. In questo modo, l'**ArrayList** originale può essere manipolato senza influenzare **notifyListeners()**.

Anche il metodo **paintComponent()** non è **synchronized**. Decidere se sincronizzare metodi sovrascritti non è una scelta così immediata come lo è per i



nuovi metodi che create. In questo esempio `paintComponent()` sembra funzionare correttamente (che sia **synchronized** oppure no), ma in ogni caso dovete tenere conto dei seguenti argomenti.

1. Il metodo modifica lo stato di variabili "critiche" all'interno dell'oggetto? Per scoprire se le variabili sono "critiche", dovete determinare se vengono lette o impostate da altri thread del programma. In tal caso, la lettura e l'impostazione vengono quasi sempre eseguite tramite i metodi **synchronized**, pertanto potrete limitarvi a esaminare questi ultimi. Nel caso di `paintComponent()` non vi è alcuna modifica.
2. Il metodo dipende dallo stato di queste variabili "critiche"? Se un metodo **synchronized** modifica una variabile utilizzata dal vostro metodo, dovreste rendere **synchronized** anche questo metodo. In base a tale considerazione, potete osservare che `cSize` è modificato da metodi **synchronized**, quindi `paintComponent()` dovrebbe essere **synchronized**. A questo punto potreste tuttavia chiedervi: "Qual è il problema più grave che potrebbe verificarsi se `cSize` venisse modificato durante l'esecuzione di `paintComponent()`?" Se ritenete che l'effetto non sia particolarmente grave, e che sia transitorio, potreste decidere di lasciare il metodo `paintComponent()` non **synchronized**, risparmiando l'onere aggiuntivo rappresentato dalla chiamata a un metodo **synchronized**.
3. Un terzo indizio consiste nell'osservare se la versione della classe di base di `paintComponent()` è **synchronized**: come vedete, non lo è. Questo non è un argomento vincolante, ma un semplice indizio. In questo caso, per esempio, un campo che è modificato tramite metodi **synchronized** (`cSize`) è stato utilizzato nel metodo `paintComponent()` e potrebbe avere cambiato la situazione. Notate, però, che **synchronized** non viene ereditato: in altre parole, se un metodo è **synchronized** nella classe di base non diviene automaticamente **synchronized** nella versione sovrascritta.
4. I metodi `paint()` e `paintComponent()` devono essere più veloci possibile. Qualsiasi intervento riduca l'onere gestionale di questi metodi è raccomandato, pertanto l'eventuale necessità di sincronizzarli potrebbe essere sintomo di errata progettazione.

Il codice di test che vedete in `main()` è stato modificato rispetto a **BangBeanTest**, per dimostrare le capacità di multicasting di **BangBean2** tramite l'aggiunta di listener supplementari.

### **Pacchettizzazione dei bean**

Prima di poter incorporare un JavaBean in un'IDE compatibile, il bean deve essere incluso in un "contenitore di bean", ossia un semplice file JAR che



include, oltre a tutte le classi che compongono il bean, il file manifesto che, come sapete, specifica trattarsi di un bean. Un file manifesto è semplicemente un file di testo conforme a una struttura definita. Per **BangBean** il file manifesto è:

```
Manifest-Version: 1.0
```

```
Name: bangbean/BangBean.class
Java-Bean: True
```

La prima riga indica la versione dello schema manifesto, che fino a nuovo avviso da parte di Sun è 1.0. Considerato che le righe vuote vengono ignorate, la seconda indica il nome **BangBean.class** e la terza riga indica semplicemente che si tratta di un bean; senza questa riga, l'IDE non riconoscerà la classe come bean.

L'unica parte che potrebbe essere fonte di errore è il campo "Name:", per il quale dovete assicurarvi di impostare correttamente il percorso. Se analizzate **BangBean.java** vedrete che è nel package **bangbean**, ovvero in una sottodirectory chiamata **bangbean** esterna al CLASSPATH; il nome nel file manifesto deve includere queste informazioni relative al package. Inoltre, dovete inserire il file manifesto nella directory *superiore* alla radice del percorso del package: in questo caso significa inserire il file nella directory che contiene la sottodirectory "bangbean". Quindi dovete eseguire **jar** dalla stessa directory contenente il file manifesto:

```
jar cfm BangBean.jar BangBean.mf bangbean
```

In questo comando si presume che il file JAR risultante sia **BangBean.jar** e che il file manifesto sia **BangBean.mf**.

Potreste domandarvi dove si trovino le altre classi generate dalla compilazione di **BangBean.java**. La risposta è semplice: queste classi sono tutte nella sottodirectory **bangbean**; infatti, l'ultimo argomento della riga di comando **jar** è la sottodirectory **bangbean**. Quando fornite a **jar** il nome di una sottodirectory, tutto il suo contenuto viene archiviato nel file JAR: in questo caso è incluso anche il codice sorgente **BangBean.java**, ma potete scegliere di non includere i sorgenti nei vostri bean. Se decomprimete il contenuto del file JAR appena creato, scoprirete che non contiene il vostro file manifesto: **jar** ha creato il proprio file manifesto chiamato **MANIFEST.MF**, parzialmente basato sul vostro, e lo ha inserito nella sottodirectory **META-INF** ("Meta



Informazioni"). Se aprite questo file, noterete che **jar** ha inserito le informazioni di firma digitale per ogni file, nella forma seguente:

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/00=
MD5-Digest: 04NcS1hE3Smnzlp2hj6qeg==
```

Generalmente, non dovete preoccuparvi di questi dati: se effettuate modifiche, potrete modificare il vostro file manifesto ed eseguire nuovamente **jar** per creare un nuovo file JAR per il vostro bean. Potete anche aggiungere altri bean al file JAR, aggiungendo le relative informazioni al file manifesto.

Probabilmente inserirete ogni bean nella propria sottodirectory: durante la creazione di un file JAR, se fornite al programma di utilità **jar** il nome di una sottodirectory tutto il contenuto verrà inserito nel file. Come potete notare, sia **Frog** sia **BangBean** vengono inseriti in sottodirectory opportune.

Quando il bean è stato inserito in un file JAR, potete utilizzarlo in un ambiente IDE compatibile con JavaBean. La tecnica da adottare è tuttavia diversa da uno strumento IDE all'altro; fortunatamente Sun fornisce un utile "banco di prova" per i JavaBeans con il suo "Bean Builder", che potete scaricare da <http://java.sun.com/beans>. Per inserire un bean in Bean Builder, è sufficiente copiare il file JAR nella sottodirectory corretta.

**Esercizio 36** (4) Aggiungete **Frog.class** al file manifesto creato in questo paragrafo, poi eseguite **jar** per creare un file JAR che contenga i bean **Frog** e **BangBean**. Procuratevi Bean Builder, o, se preferite, servitevi del vostro IDE preferito e aggiungete il file JAR all'ambiente di sviluppo in modo da testare i due bean.

**Esercizio 37** (5) Create un JavaBean chiamato **Valve** che contenga due proprietà: una proprietà di nome "on", di tipo **boolean**, e una di nome "level", di tipo **int**. Create un file manifesto, utilizzate **jar** per pacchettizzare il bean e infine caricatelo in Bean Builder o nel vostro IDE preferito per testarlo.

### **Supporto ai bean avanzato**

Avete visto quanto sia semplice creare un bean, tuttavia non limitatevi a quanto visto finora. L'architettura JavaBeans fornisce un semplice punto d'ingresso, ma può gestire situazioni anche più complesse. Sebbene tali situazioni vadano oltre la portata di questo manuale, è opportuno farvi cenno: per ulteriori dettagli fate riferimento a <http://java.sun.com/beans>.



Potete rendere più sofisticate le proprietà: gli esempi visti finora si riferiscono a proprietà singole, ma è possibile rappresentare anche proprietà multiple in un array. Questa tecnica consente di creare la cosiddetta *proprietà indicizzata (indexed property)*: dovete semplicemente fornire i metodi adatti, che naturalmente seguono una convenzione di nomenclatura, e **Introspector** riconoscerà una proprietà indicizzata affinché il vostro IDE possa rispondere adeguatamente.

Le proprietà possono essere *collegate (bound)*, indicando con questo termine che esse informeranno altri oggetti tramite un **PropertyChangeEvent**. Gli oggetti possono poi scegliere se adeguarsi, in base al cambiamento del bean.

Le proprietà possono essere *vincolate (constrained)*: altri oggetti potranno impedire la modifica di una proprietà se il nuovo valore non è considerato accettabile. Gli altri oggetti vengono informati della modifica tramite un **PropertyChangeEvent** e possono sollevare un'eccezione di tipo **PropertyVetoException** per impedire la modifica e ristabilire il valore precedente.

Potete anche modificare il modo in cui il bean è rappresentato in fase di progettazione, per esempio con il seguente procedimento.

1. Creare una finestra delle proprietà personalizzata. Tutti gli altri bean richiameranno l'elenco di proprietà standard, ma quando l'utente selezionerà il vostro bean, l'IDE richiamerà automaticamente la finestra delle proprietà personalizzata.
2. Creare un editor personalizzato per una proprietà particolare, che sarà automaticamente invocato quando l'utente modificherà la proprietà del vostro bean.
3. Fornire una classe **BeanInfo** personalizzata per il vostro bean, che produca informazioni differenti da quelle create in modo predefinito da **Introspector**.
4. Attivare o disattivare la modalità “expert” in ogni **FeatureDescriptor**, per distinguere tra le caratteristiche di base e quelle avanzate.

### ***Ulteriori risorse su JavaBeans***

Sono stati scritti numerosi titoli su JavaBeans, per esempio *JavaBeans* di Elliotte Rusty Harold (IDG, 1998).

## **Alternative a Swing**

Benché la libreria Swing sia l'interfaccia ufficiale di Sun, non è certo l'unico strumento per creare interfacce grafiche. Due alternative importanti sono *Adobe Flash* (ex Macromedia Flash), che utilizza l'ambiente di programma-



zione *Flex* di Adobe per interfacce lato client via web, e *SWT (Standard Widget Toolkit)*, la libreria open source Eclipse di IBM per applicazioni desktop.

Vi domanderete probabilmente per quale ragione dovreste considerare alcune alternative. Per i client web, un motivo importante è il fallimento degli applet. Considerate da quanti anni sono in circolazione e l'entusiasmo e le promesse che ne hanno salutato la presentazione: ciononostante, trovare un'applicazione web che utilizza gli applet è ancora una sorpresa. Neppure Sun utilizza gli applet ovunque. Osservate, per esempio, la pagina <http://java.sun.com/developer/onlineTraining/new2java/javamap/intro.html>: una mappa interattiva delle caratteristiche di Java sul sito di Sun sembra un candidato ideale per un applet Java, eppure è stata realizzata in Flash. La si direbbe una tacita ammissione dell'insuccesso degli applet. Ma più importante è il fatto che il lettore Flash Player è installato sul 98% dei computer, pertanto può essere considerato uno standard *de facto*. Come vedrete, Flex è un ambiente di programmazione lato client molto più potente di JavaScript, e con caratteristiche estetiche spesso preferibili a quelle di un applet. Se desiderate utilizzare gli applet, dovrete convincere l'utente client a scaricare JRE, al cui confronto il download di Flash Player è un'inezia.

Per le applicazioni desktop, un problema con Swing è che gli utenti potrebbero *rendersi conto* di utilizzare applicazioni di tipo diverso, qualora il loro aspetto e comportamento siano differenti da quelli dell'ambiente desktop standard. Di norma gli utenti non sono interessati a soluzioni ed estetiche alternative, e preferiscono che una nuova applicazione abbia l'aspetto e il comportamento di quelle esistenti. SWT crea applicazioni che assomigliano a quelle native, e poiché questa libreria utilizza quanti più componenti nativi del sistema è possibile, le applicazioni sono generalmente più veloci di equivalenti applicazioni Swing.

## Costruire client web in Flash con Flex

Poiché la macchina virtuale di Adobe Flash è così diffusa, la maggior parte degli utenti potrà utilizzare l'interfaccia di base Flash senza installare alcunché, e l'interfaccia avrà lo stesso aspetto e comportamento su tutti i sistemi e le piattaforme.<sup>8</sup>

Con *Adobe Flex* potete sviluppare interfacce utente Flash per le applicazioni Java. Flex è costituito da un modello di programmazione basato su XML e su script, simile a quelli che caratterizzano i linguaggi HTML e JavaScript,

8. Sean Neville ha fornito il materiale di base per la redazione di questi paragrafi.



integrato con una robusta libreria di componenti. La sintassi MXML serve per le dichiarazioni di gestione del layout e di controllo dei widget, mentre lo scripting dinamico consente di aggiungere il codice di gestione degli eventi e d'invocazione dei servizi, che collega l'interfaccia utente alle classi Java, ai modelli di dati, ai servizi web ecc. Il compilatore Flex accetta i file MXML e gli script e li compila sotto forma di bytecode. La macchina virtuale Flash sul client funziona come la JVM, in quanto interpreta il bytecode compilato. Il formato del bytecode Flash è conosciuto come SWF, e i file SWF sono prodotti dal compilatore Flex.

Noteate che esiste un'alternativa opensource a Flex, OpenLaszlo (<http://open-laszlo.org>), che pur avendo una struttura analoga a Flex sotto certi aspetti rappresenta un'alternativa preferibile. Sono disponibili anche altri strumenti per creare applicazioni Flash con tecniche diverse.

## Hello, Flex

Considerate questo codice MXML che definisce un'interfaccia utente; tenete presente che la prima e l'ultima riga non appaiono nel codice sorgente del pacchetto fornito a corredo di questo volume.

```
//:! gui/flex/helloflex1.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.macromedia.com/2003/mxml"
    backgroundColor="#ffffff">
    <mx:Label id="output" text="Hello, Flex!" />
</mx:Application>
///:~
```

I file MXML sono documenti XML, pertanto iniziano con una direttiva di versione e codifica XML. L'elemento esterno MXML è **Application**, il contenitore visivo e logico principale di un'interfaccia utente Flex. All'interno dell'elemento **Application** potete dichiarare i tag che rappresentano i comandi visivi, come l'elemento **Label** nell'esempio. I comandi vengono sempre disposti in un contenitore, e questo incapsula i gestori di layout e altri meccanismi in modo da gestire la disposizione dei comandi al suo interno. Nel caso più semplice, come l'esempio precedente, **Application** svolge il ruolo di contenitore, nel quale il layout manager predefinito di **Application** si limita a disporre i controlli verticalmente nell'interfaccia, nell'ordine in cui vengono dichiarati.



ActionScript è una versione di ECMAScript e JavaScript, che è molto simile a Java e supporta le classi, lo *strong typing* e lo scripting dinamico. Aggiungendo uno script all'esempio, potete implementare alcuni comportamenti; in questo caso, viene utilizzato il controllo **Script** per disporre delle istruzioni ActionScript direttamente nel file MXML:

```
///:! gui/flex/helloflex2.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.macromedia.com/2003/mxml"
    backgroundColor="#ffffffff">
    <mx:Script>
        <![CDATA[
            function updateOutput() {
                output.text = "Hello! " + input.text;
            }
        ]]>
    </mx:Script>
    <mx:TextInput id="input" width="200"
        change="updateOutput()" />
    <mx:Label id="output" text="Hello!" />
</mx:Application>
///:-
```

Un controllo **TextInput** accetta l'input dell'utente e una **Label** visualizza i dati via via che vengono digitati. Notate che l'attributo **id** di ogni controllo diventa accessibile nello script come nome di variabile, pertanto lo script può fare riferimento a istanze dei tag MXML. Nel campo **TextInput** potete osservare che l'attributo **change** è collegato alla funzione **updateOutput()**, in modo che la funzione possa essere chiamata ogni volta che si verifica un cambiamento.

### Compilare MXML

Il modo più facile per iniziare a utilizzare Flex è scaricare la versione dimostrativa del prodotto, disponibile all'indirizzo [www.adobe.com/products/flex/](http://www.adobe.com/products/flex/).

Il prodotto è disponibile in numerose versioni, dalle demo gratuite alle versioni server aziendali, e Adobe offre anche strumenti aggiuntivi per sviluppare applicazioni in Flex. Il contenuto dei vari pacchetti è soggetto a modifiche,

pertanto dovreste controllare direttamente sul sito di Adobe per verificare la presenza di eventuali aggiornamenti. Ricordate che, in fase d'installazione, potrete avere bisogno di modificare il file **jvm.config** nella sottodirectory **bin** della directory d'installazione di Flex.

Per compilare il codice MXML nel bytecode di Flash, avete due possibilità.

1. Disporre il file MXML in un'applicazione web Java insieme alle pagine HTML e JSP, all'interno di un file WAR, per fare in modo che il file **.mxml** sia compilato in fase di esecuzione ogni volta che il browser richiede l'URL del documento MXML.
2. Compilare il file MXML utilizzando **mxmlc**, il compilatore Flex da riga di comando.

La prima opzione, vale a dire la compilazione runtime via web, richiede anche un contenitore servlet, per esempio Apache Tomcat. Il file WAR di questo contenitore deve essere aggiornato con le informazioni di configurazione Flex, quali le mappature dei servlet che sono aggiunti al descrittore **web.xml**, e deve includere i file JAR Flex: queste operazioni vengono gestite automaticamente al momento dell'installazione di Flex. Dopo avere configurato il file WAR, potrete disporre i file MXML nell'applicazione web e richiamare l'URL del documento da qualsiasi browser. Flex compilerà l'applicazione alla prima richiesta, come avviene con il modello JSP, e da quel momento trasporterà codice SWF compilato e in cache, inglobato in una struttura HTML.

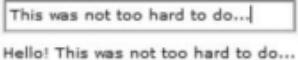
La seconda opzione non richiede un server. Quando avviate il compilatore **mxmlc** di Flex dalla riga di comando vengono prodotti file SWF, i quali potranno essere utilizzati nel modo che preferite. L'eseguibile **mxmlc** è situato nella directory **bin** del percorso di installazione di Flex, e la sua chiamata senza argomenti produrrà un elenco di opzioni. Di norma specificherete l'opzione **-flexlib**, indicando come valore la posizione della libreria di componenti client Flex: tenete presente, tuttavia, che in esempi molto semplici come quelli esaminati finora il compilatore Flex è in grado di determinare automaticamente tale posizione. Potete quindi compilare i primi due esempi nel modo seguente:

```
mxmlc.exe hellofflex1.mxml  
mxmlc.exe hellofflex2.mxml
```

Otterrete così un file **hellofflex2.swf** eseguibile in Flash, oppure da collocare insieme al codice HTML in qualsiasi server HTTP: ricordate che una volta che Flash è stato caricato nel browser, è sufficiente un doppio clic sul file SWF per attivarlo nel browser.



Eseguendo **helloflex2.swf** con Flash Player, otterrete il seguente risultato:



In caso di applicazioni più complesse, potete separare il codice MXML e ActionScript facendo riferimento alle funzioni di file ActionScript esterni. Da MXML, utilizzate la seguente sintassi per il controllo **Script**:

```
<mx:Script source="MyExternalScript.as" />
```

Questo codice fa sì che i comandi MXML possano richiamare funzioni situate in un file chiamato **MyExternalScript.as**, esattamente come se fossero presenti all'interno del file MXML.

### **MXML e ActionScript**

MXML è il linguaggio di "stenografia dichiarativa" (*declarative shorthand*) per le classi ActionScript. Ogni volta che incontrate un tag MXML, siete certi che corrisponde a un'omonima classe ActionScript. Quando il compilatore Flex analizza MXML, in primo luogo trasforma l'XML in ActionScript e carica le classi ActionScript di riferimento, poi compila e collega ("linka") l'ActionScript in un file SWF.

È possibile scrivere un'intera applicazione Flex in ActionScript senza utilizzare MXML, che rappresenta quindi soltanto un'alternativa di comodo. I componenti dell'interfaccia utente quali i contenitori e i comandi vengono solitamente dichiarati utilizzando MXML, mentre la logica applicativa, quale il gestore di eventi e l'ulteriore logica del client, viene gestita mediante ActionScript e Java.

Potete creare i vostri controlli MXML e farvi riferimento utilizzando MXML, tramite la scrittura di classi ActionScript. Potete anche combinare contenitori e controlli MXML esistenti in un nuovo documento MXML, cui potete poi fare riferimento come tag di un altro documento MXML. Sul sito web di Adobe troverete le informazioni necessarie per eseguire queste operazioni.



## Contenitori e controlli

Il nucleo visuale della libreria di componenti Flex è costituito da un insieme di contenitori che gestiscono il layout e da un array di controlli da inserire in questi contenitori. I contenitori includono pannelli, caselle orizzontali e verticali, riquadri (*tile*), caselle divise, griglie e altro; i controlli sono *widget*, quali pulsanti, caselle di testo, cursori di scorrimento, calendari, tabelle e così via.

Nei prossimi paragrafi esaminerete un'applicazione Flex che visualizza e ordina un elenco di file audio: questo programma dimostra l'utilizzo di contenitori e controlli e spiega come collegare le applicazioni Flash a Java.

Iniziate il file MXML disponendo un controllo **DataGrid**, uno dei controlli Flex più sofisticati, all'interno di un contenitore **Panel**:

```
//:! gui/flex/songs.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.macromedia.com/2003/mxml"
    backgroundColor="#B9CAD2" pageTitle="Flex Song Manager"
    initialize="getSongs()"
    <mx:Script source="songScript.as" />
    <mx:Style source="songStyles.css"/>
    <mx:Panel id="songListPanel"
        titleStyleDeclaration="headerText"
        title="Flex MP3 Library">
        <mx:HBox verticalAlign="bottom">
            <mx:DataGrid id="songGrid"
                cellPress="selectSong(event)" rowCount="8">
                <mx:columns>
                    <mx:Array>
                        <mx:DataGridColumn columnName="name"
                            headerText="Song Name" width="120" />
                        <mx:DataGridColumn columnName="artist"
                            headerText="Artist" width="180" />
                        <mx:DataGridColumn columnName="album"
                            headerText="Album" width="160" />
                    </mx:Array>
                </mx:columns>
            </mx:DataGrid>
        </mx:HBox>
    </mx:Panel>
</mx:Application>
```



```
</mx:DataGrid>
<mx:VBox>
    <mx:HBox height="100" >
        <mx:Image id="albumImage" source=""
            height="80" width="100"
            mouseOverEffect="resizeBig"
            mouseOutEffect="resizeSmall" />
        <mx:TextArea id="songInfo"
            styleName="boldText" height="100%" width="120"
            vScrollPolicy="off" borderStyle="none" />
    </mx:HBox>
    <mx:MediaPlayback id="songPlayer"
        contentPath=""
        mediaType="MP3"
        height="70"
        width="230"
        controllerPolicy="on"
        autoPlay="false"
        visible="false" />
    </mx:VBox>
</mx:HBox>
<mx:ControlBar horizontalAlign="right">
    <mx:Button id="refreshSongsButton"
        label="Refresh Songs" width="100"
        toolTip="Refresh Song List"
        click="songService.getSongs()" />
</mx:ControlBar>
</mx:Panel>
<mx:Effect>
    <mx:Resize name="resizeBig" heightTo="100"
        duration="500"/>
    <mx:Resize name="resizeSmall" heightTo="80"
        duration="500"/>
</mx:Effect>
<mx:RemoteObject id="songService"
    source="gui.flex.SongService"
```



```
result="onSongs(event.result)"  
fault="alert(event.fault.faultstring, 'Error')">>  
<mx:method name="getSongs"/>  
</mx:RemoteObject>  
</mx:Application>  
///:~
```

Il **DataGrid** contiene tag nidificati per il proprio array di colonne. Quando vedete un attributo o un elemento nidificato su un controllo, sapete che corrisponde a una certa proprietà, evento, o oggetto encapsulato nella classe ActionScript sottostante. Il **DataGrid** possiede un attributo **id** con valore **songGrid**, quindi ActionScript e i tag MXML fanno riferimento alla griglia utilizzando **songGrid** come nome di variabile. Oltre a quelle indicate in questo esempio, il **DataGrid** offre molte altre proprietà, che troverete descritte nella documentazione API completa per i controlli e i contenitori MXML, all'indirizzo [http://livedocs.macromedia.com/flex/15/asdocs\\_en/index.html](http://livedocs.macromedia.com/flex/15/asdocs_en/index.html).

Nell'esempio precedente, il **DataGrid** è seguito da una **VBox** che contiene un controllo **Image** per visualizzare la copertina dell'album con le informazioni sul brano, e da un controllo **MediaPlayback** che riproduce i file MP3. Questo esempio carica il contenuto in stream per ridurre la dimensione del file SWF compilato. Quando inserite immagini, file audio e video in un'applicazione Flex anziché caricarli in stream, i file diventano parte del file SWF compilato e vengono distribuiti insieme all'interfaccia utente. Il riproduttore Flash Player contiene i driver (*codec*) per la riproduzione e lo streaming audio e video in numerosi formati. Flash e Flex supportano i formati di immagine più comuni sul Web, e Flex ha anche la capacità di convertire i file **SVG** (*Scalable Vector Graphics*) in risorse SWF che possono essere incluse nei programmi client Flex.

### **Effetti e stili**

Flash Player renderizza la grafica servendosi dei vettori, pertanto può garantire effetti altamente espressivi in fase di esecuzione. Gli *effetti* Flex forniscono un saggio di questo tipo di animazioni, e possono essere applicati a comandi e contenitori utilizzando la sintassi MXML.

Il tag **Effect** mostrato nell'esempio MXML precedente produce due risultati: il primo tag nidificato ingrandisce dinamicamente un'immagine quando il cursore del mouse si sposta sopra di essa, e il secondo rimpicciolisce la stessa immagine quando il cursore si allontana. Questi effetti si applicano agli eventi del mouse disponibili per il controllo **Image** con l'**id albumImage**.



Flex fornisce anche effetti per animazioni comuni, quali transizioni, cancellazioni e modulazioni dei canali alfa. Oltre agli effetti incorporati, Flex supporta anche le API di disegno Flash per la creazione di animazioni innovative, la cui analisi implica argomenti come il design e l'animazione grafica che vanno oltre gli obiettivi di questo capitolo.

È anche disponibile il supporto agli stili standard CSS (*Cascading Style Sheets*): se a un file MXML viene collegato un file CSS, i controlli di Flex si adegueranno al foglio di stile. Per l'esempio **songs.mxml**, il foglio di stile **songStyles.css** contiene le seguenti dichiarazioni:

```
//!: gui/flex/songStyles.css
.headerText {
    font-family: Arial, "_sans";
    font-size: 16;
    font-weight: bold;
}

.boldText {
    font-family: Arial, "_sans";
    font-size: 11;
    font-weight: bold;
}
//:~
```

Questo file è importato e utilizzato nel gestore di brani musicali tramite il tag **Style** nel file MXML. Dopo l'importazione del foglio di stile, le sue dichiarazioni possono essere applicate ai controlli Flex nel file MXML: come esempio, la dichiarazione **boldText** presente nel foglio di stile è stata utilizzata per il controllo **TextArea** con l'**id songInfo**.

## Eventi

L'interfaccia utente è una macchina a stati che esegue azioni in risposta ai cambiamenti di stato. In Flex questi cambiamenti sono gestiti tramite gli eventi. La libreria di classi di Flex contiene un'ampia varietà di controlli, con numerosi eventi che coprono tutte le funzionalità del mouse e l'utilizzo della tastiera.

L'attributo **click** di un **Button**, per esempio, rappresenta uno degli eventi disponibili per quel controllo. Il valore assegnato a **click** può essere una fun-



zione o una porzione di script: nel file **songs.mxml**, per esempio, **ControlBar** contiene il pulsante **refreshSongsButton** che aggiorna l'elenco di brani musicali. Potete notare dal tag che quando si verifica l'evento, **click** viene chiamato **songService.getSongs()**: in questo esempio, l'evento **click** del **Button** si riferisce a **RemoteObject**, il quale corrisponde al metodo Java.

### Connetersi a Java

Il tag **RemoteObject** al termine del file MXML imposta la connessione alla classe esterna Java, **gui.flex.SongService**. Il client Flex si serve del metodo **getSongs()** nella classe Java per richiamare i dati necessari al controllo **DataGrid**; per eseguire questa operazione, deve comparire come *servizio*: un punto terminale (*endpoint*) con il quale il client può scambiare messaggi. Il servizio definito nel tag **RemoteObject** ha un attributo **source** che indica la classe Java del **RemoteObject**, e specifica una funzione callback ActionScript, **onSongs()**, che viene invocata al ritorno del metodo Java. Il tag nidificato **method** dichiara il metodo **getSongs()**, che rende il metodo Java accessibile al resto dell'applicazione Flex.

In Flex tutte le chiamate dei servizi ritornano in modo asincrono attraverso eventi attivati da queste funzioni callback; **RemoteObject** visualizza anche un controllo di dialogo per segnalare eventuali errori.

Il metodo **getSongs()** può essere invocato da Flash mediante ActionScript:

```
songService.getSongs();
```

In base alla configurazione di MXML, questa istruzione chiamerà **getSongs()** nella classe **SongService**.

```
//: gui/flex/SongService.java
package gui.flex;
import java.util.*;

public class SongService {
    private List<Song> songs = new ArrayList<Song>();
    public SongService() { fillTestData(); }
    public List<Song> getSongs() { return songs; }
    public void addSong(Song song) { songs.add(song); }
    public void removeSong(Song song) { songs.remove(song); }
    private void fillTestData() {
```



```
addSong(new Song("Chocolate", "Snow Patrol",
    "Final Straw", "sp-final-straw.jpg",
    "chocolate.mp3"));
addSong(new Song("Concerto No. 2 in E", "Hilary Hahn",
    "Bach: Violin Concertos", "hahn.jpg",
    "bachviolin2.mp3"));
addSong(new Song("Round Midnight", "Wes Montgomery",
    "The Artistry of Wes Montgomery",
    "wesmontgomery.jpg", "roundmidnight.mp3"));
}
} //:~
```

Ogni oggetto **Song** è un banale contenitore di dati:

```
//: gui/flex/Song.java
package gui.flex;

public class Song implements java.io.Serializable {
    private String name;
    private String artist;
    private String album;
    private String albumImageUrl;
    private String songMediaUrl;
    public Song() {}
    public Song(String name, String artist, String album,
        String albumImageUrl, String songMediaUrl) {
        this.name = name;
        this.artist = artist;
        this.album = album;
        this.albumImageUrl = albumImageUrl;
        this.songMediaUrl = songMediaUrl;
    }
    public void setAlbum(String album) { this.album = album; }
    public String getAlbum() { return album; }
    public void setAlbumImageUrl(String albumImageUrl) {
        this.albumImageUrl = albumImageUrl;
    }
}
```



```
public String getAlbumImageUrl() { return albumImageUrl; }
public void setArtist(String artist) {
    this.artist = artist;
}
public String getArtist() { return artist; }
public void setName(String name) { this.name = name; }
public String getName() { return name; }
public void setSongMediaUrl(String songMediaUrl) {
    this.songMediaUrl = songMediaUrl;
}
public String getSongMediaUrl() { return songMediaUrl; }
}///:~
```

Quando l'applicazione viene inizializzata, o quando si preme il pulsante **refreshSongsButton**, viene chiamato il metodo **getSongs()**; al suo ritorno viene chiamato l'ActionScript **onSongs(event.result)**, che popola il controllo **songGrid**.

Ecco il listato ActionScript, incluso con il controllo **Script** nel file MXML:

```
//: gui/flex/songScript.as
function getSongs() {
    songService.getSongs();
}

function selectSong(event) {
    var song = songGrid.getItemAt(event.itemIndex);
    showSongInfo(song);
}

function showSongInfo(song) {
    songInfo.text = song.name + newline;
    songInfo.text += song.artist + newline;
    songInfo.text += song.album + newline;
    albumImage.source = song.albumImageUrl;
    songPlayer.contentPath = song.songMediaUrl;
    songPlayer.visible = true;
}
```



```
function onSongs(songs) {  
    songGrid.dataProvider = songs;  
} //:/~
```

Per gestire la selezione delle celle del controllo **DataGrid**, alla dichiarazione **DataGrid** nel file MXML viene aggiunto l'attributo di evento **cellPress**:

```
cellPress="selectSong(event)"
```

Quando l'utente fa clic su un brano contenuto nel **DataGrid**, questo chiamarà **selectSong()** nell'ActionScript descritto in precedenza.

### **Modelli dei dati e collegamento ai dati**

I comandi possono invocare direttamente i servizi, pertanto le callback di evento ActionScript consentono di aggiornare i comandi visuali quando i servizi restituiscono i dati.

Sebbene lo script che aggiorna i comandi sia semplice e diretto, in alcuni casi può diventare prolioso e confuso; in ogni caso, le sue funzionalità sono così comuni che Flex gestisce automaticamente il comportamento per il collegamento ai dati (*data binding*).

Nella sua forma più semplice, questo collegamento fa sì che i controlli si riferiscono direttamente ai dati, anziché ricorrere alla copia diretta dei dati in un controllo. Il controllo di riferimento viene automaticamente aggiornato a ogni aggiornamento dei dati: l'infrastruttura di Flex risponde correttamente agli eventi di modifica dei dati e aggiorna tutti i controlli che sono collegati ai dati.

Ecco un semplice esempio della sintassi di collegamento ai dati:

```
<mx:Slider id="mySlider"/>  
<mx:Text text="{mySlider.value}">
```

Per realizzare il data binding, disponete i riferimenti all'interno di parentesi graffe: {}. Tutto ciò che si trova all'interno di queste parentesi è considerato un'espressione che Flex dovrà valutare.

Il valore del primo controllo, un widget **Slider**, è visualizzato tramite il secondo controllo, un campo **Text**; mentre **Slider** cambia, la proprietà **text** del campo **Text** viene automaticamente aggiornata: così facendo, lo sviluppatore non deve gestire gli eventi di modifica dello **Slider** per aggiornare il campo **Text**.

Alcuni controlli (quali il controllo **Tree** e il **DataGrid** nell'applicazione della libreria di brani musicali) sono più elaborati, e dispongono di una proprietà **dataprovider** che semplifica il collegamento a raccolte di dati.

La funzione ActionScript **onSongs()** mostra come collegare il metodo **SongService.getSongs()** al **dataprovider** del **DataGrid** di Flex; come dichiarato nel tag **RemoteObject** del file MXML, questa funzione rappresenta il callback chiamato da ActionScript ogni volta che il metodo Java ritorna.

Progetti più specializzati con una gestione dati più articolata, quali possono essere un'applicazione aziendale che utilizza DTO (*Data Transfer Object*) o un programma di messaggistica con dati strutturati in schemi complessi, potrebbero giustificare l'ulteriore separazione della sorgente dati dai controlli. Nello sviluppo Flex, tale separazione viene realizzata dichiarando un oggetto "Model", un contenitore generico MXML per i dati.

Il modello non contiene logica, ma si limita a riflettere il DTO implementato nell'applicazione aziendale e le strutture di altri linguaggi di programmazione. Servendovi di questo modello potete collegare i dati dei controlli al modello, e nello stesso tempo fare in modo che il modello colleghi i dati delle proprietà alle funzionalità di input e output del servizio.

In questo modo, le sorgenti dei dati, ossia i servizi, risultano separate dai "consumatori visuali" dei dati, vale a dire i controlli, semplificando l'applicazione del modello MVC (*Model-View-Controller*). In programmi specialistici di grandi dimensioni, la complessità iniziale causata dall'implementazione di un modello è spesso un piccolo prezzo da pagare in rapporto ai benefici che la netta separazione del modello MVC è in grado di offrire.

Oltre agli oggetti Java, Flex può anche accedere ai servizi web di tipo SOAP e ai servizi HTTP RESTful, utilizzando rispettivamente i comandi **HttpService** e **WebService**. L'accesso a tutti i servizi è soggetto ad autorizzazioni di sicurezza.

### **Compilazione e distribuzione**

Negli esempi iniziali la compilazione è stata possibile anche senza il flag **-flexlib** da riga di comando; tuttavia, per compilare il programma di gestione musicale, dovete specificare la posizione del file **flex-config.xml** utilizzando il flag **-flexlib**.

Sul computer dell'autore ha funzionato perfettamente la riga di comando indicata di seguito, che dovrete però modificare in funzione della vostra config-



gurazione; tenete presente che il comando consiste di un'unica riga, spezzata per ovvi motivi di formattazione:

```
//:! gui/flex/build-command.txt
mxmlc -flexlib C:/Program Files/Macromedia/Flex/jrun4/
servers/default/flex/WEB-INF/flex songs.mxml
//:~
```

Questo comando compilerà l'applicazione in un file SWF, che potrete visualizzare nel vostro browser; il file contenente il codice di questo volume, però, non contiene né file MP3 né file JPG, e di conseguenza eseguendo l'applicazione ne vedrete soltanto la struttura.

Inoltre, per comunicare con i file Java dall'applicazione Flex dovete configurare un server. Nel pacchetto dimostrativo di Flex è fornito il server JRun, che potrete avviare dal menu di Flex oppure direttamente dalla riga di comando:

```
jrun -start default
```

Per verificare se il server è stato avviato con successo, digitate l'URL *http://localhost:8700/samples* in un browser qualsiasi e provate i vari campioni presenti nella pagina; questo è peraltro un modo eccellente per familiarizzare con le funzionalità di Flex.

Anziché compilare l'applicazione dalla riga di comando, potete farlo tramite il server. A questo scopo, inserite i file sorgenti, i file audio, il foglio di stile ecc. nella directory **jrun4/servers/default/flex** e accedetevi dal browser digitando *http://localhost:8700/flex/songs.mxml*. Per eseguire l'applicazione dovete configurare sia il lato Java sia il lato di Flex, come indicato di seguito.

**Java:** i file compilati **SongService.java** e **Song.java** devono essere disposti nella directory **WEB-INF/classes**. Secondo le specifiche J2EE, questa è la posizione in cui inserire le classi WAR; in alternativa, potete archiviare i file con JAR e trasferire il risultato in **WEB-INF/lib**. I file compilati Java dovranno essere ospitati in una directory che replica la struttura del relativo pacchetto Java. Se state utilizzando JRun, questi file compilati dovrebbero essere posizionati in **jrun4/servers/default/flex/WEB-INF/classes/gui/flex/Song.class** e **jrun4/servers/default/flex/WEB-INF/classes/gui/flex/SongService.class**. Naturalmente, anche i file per le immagini e i file MP3 dovranno essere disponibili all'applicazione web; tenete presente che, per JRun, **jrun4/servers/default/flex** è la radice in cui risiede l'applicazione web.



**Flex:** per motivi di sicurezza Flex non può accedere agli oggetti Java, a meno che non venga autorizzato mediante l'opportuna configurazione del file **flex-config.xml**. Per JRun, il percorso di questo file è **jrun4/servers/default/flex/WEB-INF/flex/flex-config.xml**. All'interno della voce **<remote-objects>** di questo file, in corrispondenza della sezione **<whitelist>** troverete la seguente nota:

*<!--*

*For security, the whitelist is locked down by default. Uncomment the source element below to enable access to all classes during development.*

*We strongly recommend not allowing access to all source files in production, since this exposes Java and Flex system classes.*

**<source>\*</source>**

**-->**

*(<!--*

*Per motivi di sicurezza, in modalità predefinita la whitelist è disattivata. Eliminate i commenti dall'elemento <source>\*</source> qui sotto per abilitare l'accesso a tutte le classi in fase di sviluppo.*

*Si raccomanda vivamente di non consentire l'accesso a tutti i file sorgente nella versione di produzione, poiché questo rende disponibili le classi di sistema Flex e Java.*

**<source>\*</source>**

**--> )**

Eliminate i commenti da questa voce per consentire l'accesso, facendo in modo che diventi **<source>\*</source>**. Il significato di questa e di altre voci di configurazione è descritto nella documentazione Flex.

**Esercizio 38 (3)** Sviluppate il “semplice esempio della sintassi di collegamento ai dati” del paragrafo “Modelli dei dati e collegamento ai dati”.

**Esercizio 39 (4)** Il codice scaricabile per questo volume non include i file MP3 e JPG indicati nel codice di **SongService.java**. Procuratevi alcuni file di questo tipo, modificate **SongService.java** per includere i loro nomi di file, quindi scaricate la versione di prova di Flex e compilate l'applicazione.



## Creare applicazioni SWT

Come si è detto, Swing ha adottato l'approccio di costruire tutti i componenti dell'interfaccia utente pezzo per pezzo, per fornire ogni componente necessario a prescindere che il sistema operativo ospite lo possieda. SWT si pone nel mezzo, utilizzando i componenti nativi se il sistema operativo li prevede, e generandoli *ex novo* in caso contrario. Ne risulta un'applicazione che all'utente appare come nativa, e che spesso ha prestazioni notevolmente superiori a un programma equivalente realizzato con Swing. Inoltre, il modello di programmazione SWT è spesso meno macchinoso di Swing, caratteristica sicuramente desiderabile in gran parte delle applicazioni.<sup>9</sup>

Poiché SWT utilizza il sistema operativo nativo per eseguire quante più operazioni possibile, può automaticamente trarre profitto dalle caratteristiche che potrebbero non essere direttamente disponibili a Swing: per esempio, Windows offre la funzionalità di "subpixel rendering" che rende i font più leggibili sugli schermi LCD.

Servendosi di SWT è anche possibile creare gli applet.

Questo paragrafo non vuole essere un'introduzione completa a SWT, ma semplicemente fornirvi informazioni sufficienti per mostrarvi come SWT si rapporta a Swing. Scoprirete che esistono numerosi widget SWT, tutti piuttosto facili da utilizzare, di cui potrete esplorare i dettagli nella ricca documentazione e nei numerosi esempi presenti sul sito [www.eclipse.org](http://www.eclipse.org). Sono disponibili anche alcuni manuali di programmazione dedicati a SWT, e altri sono in preparazione.

### Installazione di SWT

Per eseguire le applicazioni SWT, dovete scaricare e installare la libreria SWT del progetto Eclipse. Collegatevi a [www.eclipse.org/downloads/](http://www.eclipse.org/downloads/), poi selezionate *Standard Widget Toolkit(SWT)*; nella pagina successiva, selezionate l'architettura del vostro sistema operativo (Windows, Linux ecc.) e fate clic sul collegamento per scaricare una copia del software: vi sarà chiesto di scegliere il mirror che preferite. All'interno del file scaricato, troverete il file **swt.jar**.

Il metodo più semplice per installare il file **swt.jar** consiste nell'inserirlo nella directory **jre/lib/ext**, in modo da non dovere apportare modifiche al CLASS-

<sup>9</sup>. Chris Grindstaff è stato di grande aiuto nella conversione degli esempi in SWT e nel fornire le necessarie informazioni sulla libreria SWT.



PATH. Quando decomprimete il file contenente la libreria SWT, troverete altri file che dovrete installare nelle posizioni opportune, a seconda del sistema operativo.

Per esempio, la distribuzione Win32 viene fornita con alcuni file DLL che devono essere inseriti nella directory **java.library.path**: di solito questa variabile d'ambiente ha lo stesso valore della variabile d'ambiente PATH, ma è opportuno che eseguiate il programma **object>ShowProperties.java** per accertarvene. Una volta eseguita questa operazione, potrete compilare ed eseguire un'applicazione SWT come se fosse un normale programma Java.

La documentazione di SWT è disponibile in un file scaricabile a parte.

Un metodo alternativo per installare SWT prevede l'installazione dell'editor Eclipse, che include sia SWT sia la relativa documentazione, facilmente consultabile tramite l'aiuto in linea di Eclipse.

## **Hello, SWT**

Inizierete con l'applicazione più semplice possibile, nel classico stile "ciao, mondo!":

```
//: swt/HelloSWT.java
// {Requires: org.eclipse.swt.widgets.Display; dovete
// installare la libreria SWT da http://www.eclipse.org }
import org.eclipse.swt.widgets.*;

public class HelloSWT {
    public static void main(String [] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Hi there, SWT!"); // Barra del titolo
        shell.open();
        while(!shell.isDisposed())
            if(!display.readAndDispatch())
                display.sleep();
        display.dispose();
    }
} ///:-
```



Se avete scaricato il codice sorgente di questo manuale, vedrete che il comando “Requires” nel file Ant `build.xml` indica i prerequisiti per la compilazione della directory `swt`: tutti i file che importano `org.eclipse.swt` richiedono l’installazione della libreria SWT di [www.eclipse.org](http://www.eclipse.org).

La classe `Display` gestisce il collegamento che “fa da ponte” (*Bridge*) tra SWT e il sistema operativo. `Shell` è invece la finestra principale, all’interno della quale sono sviluppati tutti gli altri componenti. Chiamando il metodo `setText()`, viene impostato il testo contenuto nella barra del titolo.

Per visualizzare la finestra e l’applicazione dovete chiamare il metodo `open()` sull’oggetto `Shell`.

Mentre Swing vi nasconde il ciclo di gestione degli eventi, SWT vi costringe a scriverli esplicitamente. Nella parte superiore del ciclo `while` verificate se la finestra principale è stata visualizzata: notate che questo vi offre la possibilità di inserire codice per eseguire eventuali attività di cleanup. Questo significa che il thread `main()` è il thread dell’interfaccia utente; in effetti, mentre dentro le quinte in Swing viene creato un secondo thread di trasmissione degli eventi, in SWT è il thread `main()` che gestisce l’interfaccia utente. Poiché in modalità predefinita esiste un solo thread e non due, questo approccio rende meno “affollata” di thread l’interfaccia grafica.

Non dovete preoccuparvi dell’invio dei task al thread dell’interfaccia utente, come fareste in Swing. Non solo SWT si prende cura di questo dettaglio per voi, ma solleva un’eccezione se cercate di gestire un widget con il thread errato. Tuttavia, al fine di creare altri thread per eseguire operazioni di lunga durata dovete sottoporre le modifiche in modo analogo a quanto avviene in Swing. Per questo SWT fornisce tre metodi che possono essere chiamati sull’oggetto `Display`: `asynExec(Runnable)`, `syncExec(Runnable)` e `timerExec(int, Runnable)`.

A questo punto, l’attività del vostro thread `main()` consiste semplicemente nel chiamare `readAndDispatch()` sull’oggetto `Display`: questo significa che può esistere un solo oggetto `Display` per ogni applicazione. Il metodo `readAndDispatch()` restituisce `true` se la coda di eventi contiene più eventi in attesa di essere elaborati, e in tal caso li richiamerà immediatamente. Se non vi sono eventi in attesa, invece, viene chiamato il metodo `sleep()` dell’oggetto `Display` e il ciclo `while` attende qualche istante (`sleep()`) prima di controllare nuovamente la coda di eventi.

Al termine del programma, dovete chiamare esplicitamente il metodo `dispose()` sull’oggetto `Display`: questo fa sì che SWT metta a disposizione le risorse, in quanto esse vengono solitamente fornite dal sistema operativo, il quale potrebbe, a sua volta, averle esaurite.



Per dimostrare che **Shell** è la finestra principale, il programma seguente crea un certo numero di oggetti **Shell**:

```
//: swt/ShellsAreMainWindows.java
import org.eclipse.swt.widgets.*;

public class ShellsAreMainWindows {
    static Shell[] shells = new Shell[10];
    public static void main(String [] args) {
        Display display = new Display();
        for(int i = 0; i < shells.length; i++) {
            shells[i] = new Shell(display);
            shells[i].setText("Shell #" + i);
            shells[i].open();
        }
        while(!shellsDisposed())
            if(!display.readAndDispatch())
                display.sleep();
        display.dispose();
    }
    static boolean shellsDisposed() {
        for(int i = 0; i < shells.length; i++)
            if(shells[i].isDisposed())
                return true;
        return false;
    }
} ///:-
```

Eseguendo questo codice si apriranno dieci finestre principali. Per il modo in cui è strutturato questo programma, chiudendo una finestra qualsiasi verranno chiuse anche quelle restanti.

Anche SWT si serve dei layout manager, in modo diverso da Swing ma applicando concetti simili. L'esempio che segue, più complesso, acquisisce testo tramite **System.getProperties()** e lo aggiunge all'oggetto shell:

```
//: swt/DisplayProperties.java
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
```



```
import org.eclipse.swt.layout.*;
import java.io.*;

public class DisplayProperties {
    public static void main(String [] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Display Properties");
        shell.setLayout(new FillLayout());
        Text text = new Text(shell, SWT.WRAP | SWT.V_SCROLL);
        StringWriter props = new StringWriter();
        System.getProperties().list(new PrintWriter(props));
        text.setText(props.toString());
        shell.open();
        while(!shell.isDisposed())
            if(!display.readAndDispatch())
                display.sleep();
        display.dispose();
    }
} ///:~
```

In SWT tutti i widget devono avere un oggetto genitore del tipo generale **Composite**, e questo genitore deve essere fornito come primo argomento al costruttore del widget. Questa operazione è eseguita nel costruttore **Text**, in cui **shell** è il primo argomento. Quasi tutti i costruttori accettano come argomento anche un flag che vi consente di specificare alcune direttive di stile, secondo quanto previsto dal widget in questione. Direttive di stile multiple sono soggette a un'operazione OR bit a bit, come vedete nell'esempio (**...SWT.WRAP | SWT.V\_SCROLL**).

Durante l'impostazione dell'oggetto **Text()** l'autore ha aggiunto i flag di stile, in modo che il testo, all'occorrenza, fosse mandato a capo (**SWT.WRAP**) e venisse aggiunta la barra di scorrimento verticale (**SWT.V\_SCROLL**). Scoprirete che SWT è intimamente legato al costruttore; esistono numerosi attributi di un widget che sono difficili o impossibili da modificare, se non mediante il costruttore. Consultate sempre la documentazione del widget per verificare quali flag sono accettati; tenete presente che alcuni costruttori richiedono un argomento flag anche quando la documentazione non elenca i flag "accettati"; questo accorgimento consente espansioni future senza modificare l'interfaccia.



## Eliminare il codice ridondante

Prima di procedere, dovete sapere che determinate operazioni devono essere eseguite per ogni applicazione SWT, proprio come accade con i programmi Swing, che prevedono azioni ripetitive. In SWT dovete sempre creare un **Display**, un oggetto **Shell** dal **Display**, poi creare un ciclo **readAndDispatch()** e così via. Naturalmente in casi particolari potrebbe non essere così, ma questa ridondanza di operazioni è così comune che è sempre opportuno cercare di eliminare il codice duplicato, in modo analogo a quanto fatto con **net.mindview.util.SwingConsole**.

Dovete fare in modo che ogni applicazione sia conforme a un'interfaccia:

```
//: swt/util/SWTApplication.java
package swt.util;
import org.eclipse.swt.widgets.*;

public interface SWTApplication {
    void createContents(Composite parent);
} //://:-
```

All'applicazione viene passato un oggetto **Composite**, di cui **Shell** è una sottoclasse, che deve essere utilizzato all'interno di **createContents()** per creare il contenuto della finestra.

Il metodo **SWTConsole.run()** chiama **createContents()** nel punto appropriato, imposta le dimensioni della finestra principale shell secondo i parametri forniti dall'utente al metodo **run()**, poi apre la shell ed esegue il ciclo di eventi, liberando infine le risorse utilizzate dalla shell all'uscita del programma:

```
//: swt/util/SWTConsole.java
package swt.util;
import org.eclipse.swt.widgets.*;

public class SWTConsole {
    public static void
    run(SWTApplication swtApp, int width, int height) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText(swtApp.getClass().getSimpleName());
```



```
swtApp.createContents(shell);
shell.setSize(width, height);
shell.open();
while(!shell.isDisposed()) {
    if(display.readAndDispatch())
        display.sleep();
}
display.dispose();
}
} //:~
```

Inoltre, al titolo della finestra viene assegnato il nome della classe **SWTApplication** e vengono impostate la larghezza (**width**) e l'altezza (**height**) della finestra **Shell**.

È possibile creare una variante di **DisplayProperties.java** che visualizzi le variabili d'ambiente del sistema, servendosi di **SWTConsole**:

```
//: swt/DisplayEnvironment.java
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.util.*;
public class DisplayEnvironment implements SWTApplication {
    public void createContents(Composite parent) {
        parent.setLayout(new FillLayout());
        Text text = new Text(parent, SWT.WRAP | SWT.V_SCROLL);
        for(Map.Entry entry: System.getenv().entrySet()) {
            text.append(entry.getKey() + ":" + " "
                + entry.getValue() + "\n");
        }
    }
    public static void main(String [] args) {
        SWTConsole.run(new DisplayEnvironment(), 800, 600);
    }
} //:~
```



**SWTConsole** vi permette quindi di concentrarvi sull'aspetto applicativo, anziché sul codice ripetitivo.

**Esercizio 40** (4) Modificate **DisplayProperties.java** in modo che utilizzi **SWTConsole**.

**Esercizio 41** (4) Modificate **DisplayEnvironment.java** in modo che *non* utilizzi **SWTConsole**.

### Menu

Per dimostrare il funzionamento di un menu di base, l'esempio seguente legge il proprio codice sorgente e lo classifica nelle singole parole che lo compongono, poi utilizzate per compilare un menu:

```
//: swt/Menus.java
// Divertirsi con i menu.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import java.util.*;
import net.mindview.util.*;

public class Menus implements SWTApplication {
    private static Shell shell;
    public void createContents(Composite parent) {
        shell = parent.getShell();
        Menu bar = new Menu(shell, SWT.BAR);
        shell.setMenuBar(bar);
        Set<String> words = new TreeSet<String>(
            new TextFile("Menus.java", "\\\W+"));
        Iterator<String> it = words.iterator();
        while(it.next().matches("[0-9]+"))
            ; // Si sposta dopo i numeri.
        MenuItem[] mItem = new MenuItem[7];
        for(int i = 0; i < mItem.length; i++) {
            mItem[i] = new MenuItem(bar, SWT.CASCADE);
            mItem[i].setText(it.next());
            Menu submenu = new Menu(shell, SWT.DROP_DOWN);
```



```
    mItem[i].setMenu(submenu);
}
int i = 0;
while(it.hasNext()) {
    addItem(bar, it, mItem[i]);
    i = (i + 1) % mItem.length;
}
}
static Listener listener = new Listener() {
    public void handleEvent(Event e) {
        System.out.println(e.toString());
    }
};
void
addItem(Menu bar, Iterator<String> it, MenuItem mItem) {
    MenuItem item = new MenuItem(mItem.getMenu(), SWT.PUSH);
    item.addListener(SWT.Selection, listener);
    item.setText(it.next());
}
public static void main(String[] args) {
    SWTConsole.run(new Menus(), 600, 200);
}
} //:~
```

Un **Menu** deve essere posto in una finestra **Shell**, e **Composite** vi permette di recuperarne la shell con **getShell()**. La classe **TextFile** proviene dalla libreria **net.mindview.util**, descritta in precedenza; qui un oggetto **TreeSet** viene popolato con le parole, che appariranno in ordine alfabetico; gli elementi iniziali, in quanto numeri, vengono scartati. Utilizzando il flusso di parole viene assegnato un nome ai menu di alto livello, poi sono creati i sottomenu a cui vengono assegnate, come nomi, tutte le parole rimaste.

In risposta alla selezione di una delle voci di menu, il **Listener** si limita a visualizzare l'evento, in modo da evidenziare il tipo di informazioni contenute nell'evento stesso. Quando eseguite il programma vedrete che tali informazioni includono l'etichetta del menu, che potrete utilizzare come riferimento per altri tipi di risposta dal menu; in alternativa potete fornire un listener diverso per ogni menu, tecnica più sicura e consigliata in caso di internazionalizzazione.



## Pannelli a schede, pulsanti ed eventi

SWT ha un ricco insieme di comandi, chiamati *widget*: la documentazione per `org.eclipse.swt.widgets` illustra i widget di base, mentre quella per `org.eclipse.swt.custom` si riferisce ai widget più caratteristici.

Per dimostrare alcuni widget di base, questo esempio dispone un certo numero di sottoesempi all'interno di pannelli a schede; inoltre, mostra come creare i **Composite** (del tutto analoghi ai **JPanels** di Swing) per includere elementi in altri elementi.

```
//: swt/TabbedPane.java
// Come inserire dei componenti SWT in pannelli a schede.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.browser.*;

public class TabbedPane implements SWTApplication {
    private static TabFolder folder;
    private static Shell shell;
    public void createContents(Composite parent) {
        shell = parent.getShell();
        parent.setLayout(new FillLayout());
        folder = new TabFolder(shell, SWT.BORDER);
        labelTab();
        directoryDialogTab();
        buttonTab();
        sliderTab();
        scribbleTab();
        browserTab();
    }
    public static void labelTab() {
        TabItem tab = new TabItem(folder, SWT.CLOSE);
        tab.setText("A Label"); // Testo sulla scheda
        tab.setToolTipText("A simple label");
    }
}
```



```
Label label = new Label(folder, SWT.CENTER);
label.setText("Label text");
tab.setControl(label);
}
public static void directoryDialogTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Directory Dialog");
    tab.setToolTipText("Select a directory");
    final Button b = new Button(folder, SWT.PUSH);
    b.setText("Select a Directory");
    b.addListener(SWT.MouseDown, new Listener() {
        public void handleEvent(Event e) {
            DirectoryDialog dd = new DirectoryDialog(shell);
            String path = dd.open();
            if(path != null)
                b.setText(path);
        }
    });
    tab.setControl(b);
}
public static void buttonTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Buttons");
    tab.setToolTipText("Different kinds of Buttons");
    Composite composite = new Composite(folder, SWT.NONE);
    composite.setLayout(new GridLayout(4, true));
    for(int dir : new int[]{SWT.UP, SWT.RIGHT, SWT.LEFT, SWT.DOWN
    }) {
        Button b = new Button(composite, SWT.ARROW | dir);
        b.addListener(SWT.MouseDown, listener);
    }
    newButton(composite, SWT.CHECK, "Check button");
    newButton(composite, SWT.PUSH, "Push button");
    newButton(composite, SWT.RADIO, "Radio button");
    newButton(composite, SWT.TOGGLE, "Toggle button");
}
```



```
    newButton(composite, SWT.FLAT, "Flat button");
    tab.setControl(composite);
}
private static Listener listener = new Listener() {
    public void handleEvent(Event e) {
        MessageBox m = new MessageBox(shell, SWT.OK);
        m.setMessage(e.toString());
        m.open();
    }
};
private static void newButton(Composite composite,
    int type, String label) {
    Button b = new Button(composite, type);
    b.setText(label);
    b.addListener(SWT.MouseDown, listener);
}
public static void sliderTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Sliders and Progress bars");
    tab.setToolTipText("Tied Slider to ProgressBar");
    Composite composite = new Composite(folder, SWT.NONE);
    composite.setLayout(new GridLayout(2, true));
    final Slider slider =
        new Slider(composite, SWT.HORIZONTAL);
    final ProgressBar progress =
        new ProgressBar(composite, SWT.HORIZONTAL);
    slider.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent event) {
            progress.setSelection(slider.getSelection());
        }
    });
    tab.setControl(composite);
}
public static void scribbleTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Scribble");
```



```
tab.setToolTipText("Simple graphics: drawing");
final Canvas canvas = new Canvas(folder, SWT.NONE);
ScribbleMouseListener sml= new ScribbleMouseListener();
canvas.addMouseListener(sml);
canvas.addMouseMoveListener(sml);
tab.setControl(canvas);
}
private static class ScribbleMouseListener
extends MouseAdapter implements MouseMoveListener {
private Point p = new Point(0, 0);
public void mouseMove(MouseEvent e) {
if((e.stateMask & SWT.BUTTON1) == 0)
return;
GC gc = new GC((Canvas)e.widget);
gc.drawLine(p.x, p.y, e.x, e.y);
gc.dispose();
updatePoint(e);
}
public void mouseDown(MouseEvent e) { updatePoint(e); }
private void updatePoint(MouseEvent e) {
p.x = e.x;
p.y = e.y;
}
}
public static void browserTab() {
TabItem tab = new TabItem(folder, SWT.CLOSE);
tab.setText("A Browser");
tab.setToolTipText("A Web Browser");
Browser browser = null;
try {
browser = new Browser(folder, SWT.NONE);
} catch(SWTError e) {
Label label = new Label(folder, SWT.BORDER);
label.setText("Could not initialize browser");
tab.setControl(label);
}
if(browser != null) {
```



```

        browser.setUrl("http://www.mindview.net");
        tab.setControl(browser);
    }
}
public static void main(String[] args) {
    SWTConsole.run(new TabbedPane(), 800, 600);
}
} //:-

```

In questo codice il metodo **createContents()** imposta il layout, poi chiama i metodi, ciascuno dei quali crea una scheda diversa. Il testo di ogni scheda è definito con **setText()**: tenete presente che potete anche creare pulsanti e immagini. A ogni scheda è associato anche un tooltip. Al termine di ogni metodo, la chiamata a **setControl()** dispone il controllo creato dal metodo nello spazio di dialogo della scheda in questione.

Il metodo **labelTab()** genera una semplice etichetta di testo, mentre **directoryDialogTab()** crea un pulsante che apre un oggetto **DirectoryDialog** standard, in modo che l'utente possa selezionare una directory: il nome della directory scelta viene utilizzato come testo del pulsante.

Il metodo **buttonTab()** mostra i vari pulsanti di base; **sliderTab()** riproduce l'esempio Swing di collegare un cursore scorrevole a una barra di avanzamento.

Il metodo **scribbleTab()** è un interessante esempio di grafica: un programma di disegno realizzato con poche righe di codice.

Per concludere, **browserTab()** dimostra la potenza del componente **Browser** di SWT, un vero e proprio browser racchiuso in un solo componente.

## Grafica

L'esempio seguente propone il programma Swing **SineWave.java** convertito in SWT:

```

//: swt/SineWave.java
// Trasposizione in SWT del programma Swing SineWave.java.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;

```



```
class SineDraw extends Canvas {  
    private static final int SCALEFACTOR = 200;  
    private int cycles;  
    private int points;  
    private double[] sines;  
    private int[] pts;  
    public SineDraw(Composite parent, int style) {  
        super(parent, style);  
        addPaintListener(new PaintListener() {  
            public void paintControl(PaintEvent e) {  
                int maxWidth = getSize().x;  
                double hstep = (double)maxWidth / (double)points;  
                int maxHeight = getSize().y;  
                pts = new int[points];  
                for(int i = 0; i < points; i++)  
                    pts[i] = (int)((sines[i] * maxHeight / 2 * .95)  
                        + (maxHeight / 2));  
                e.gc.setForeground(  
                    e.display.getSystemColor(SWT.COLOR_RED));  
                for(int i = 1; i < points; i++) {  
                    int x1 = (int)((i - 1) * hstep);  
                    int x2 = (int)(i * hstep);  
                    int y1 = pts[i - 1];  
                    int y2 = pts[i];  
                    e.gc.drawLine(x1, y1, x2, y2);  
                }  
            }  
        });  
        setCycles(5);  
    }  
    public void setCycles(int newCycles) {  
        cycles = newCycles;  
        points = SCALEFACTOR * cycles * 2;  
        sines = new double[points];  
        for(int i = 0; i < points; i++) {  
            double radians = (Math.PI / SCALEFACTOR) * i;
```



```
sines[i] = Math.sin(radians);
}
redraw();
}
}

public class SineWave implements SWTApplication {
    private SineDraw sines;
    private Slider slider;
    public void createContents(Composite parent) {
        parent.setLayout(new GridLayout(1, true));
        sines = new SineDraw(parent, SWT.NONE);
        sines.setLayoutData(
            new GridData(SWT.FILL, SWT.FILL, true, true));
        sines.setFocus();
        slider = new Slider(parent, SWT.HORIZONTAL);
        slider.setValues(5, 1, 30, 1, 1, 1);
        slider.setLayoutData(
            new GridData(SWT.FILL, SWT.DEFAULT, true, false));
        slider.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                sines.setCycles(slider.getSelection());
            }
        });
    }
    public static void main(String[] args) {
        SWTConsole.run(new SineWave(), 700, 400);
    }
} //:~
```

In questo caso, invece del componente **JPanel**, l'area di disegno principale di SWT è **Canvas**.

Se paragonate questa versione del programma all'originale Swing, noterete che la classe **SineDraw** è sostanzialmente identica. In SWT il contesto grafico **gc** si ottiene dall'oggetto evento che viene passato a **PaintListener**, mentre in Swing l'oggetto **Graphics** è passato direttamente al metodo **paintComponent()**. In entrambi i casi, le attività eseguite con gli oggetti grafici sono le stesse e il metodo **setCycles()** è identico.



Il metodo `createContents()` richiede ancor meno codice della versione Swing per impostare la finestra, il cursore scorrevole e il relativo listener, e anche in questo caso le attività di base sono pressoché le stesse.

### Concorrenza in SWT

Anche se AWT/Swing è a thread singolo, non è difficile contravvenire a queste regole e creare un programma non deterministico. In generale, però, non è opportuno avere thread multipli che scrivono sulla finestra, perché potrebbero interagire in maniera inaspettata.

SWT non ammette questa tecnica, e solleva un'eccezione nel caso tentiate di scrivere sulla finestra servendovi di più di un thread. In tal modo al programmatore debuttante viene impedito di compiere questo errore, che potrebbe introdurre bug molto complessi da individuare. L'esempio seguente è la traduzione in SWT del programma Swing `ColorBoxes.java`:

```
//: swt/ColorBoxes.java
// Trasposizione in SWT del programma Swing ColorBoxes.java.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class CBox extends Canvas implements Runnable {
    class CBoxPaintListener implements PaintListener {
        public void paintControl(PaintEvent e) {
            Color color = new Color(e.display, cColor);
            e.gc.setBackground(color);
            Point size = getSize();
            e.gc.fillRect(0, 0, size.x, size.y);
            color.dispose();
        }
    }
}
```



```
private static Random rand = new Random();
private static RGB newColor() {
    return new RGB(rand.nextInt(255),
        rand.nextInt(255), rand.nextInt(255));
}
private int pause;
private RGB cColor = newColor();
public CBox(Composite parent, int pause) {
    super(parent, SWT.NONE);
    this.pause = pause;
    addPaintListener(new CBoxPaintListener());
}
public void run() {
    try {
        while(!Thread.interrupted()) {
            cColor = newColor();
            getDisplay().asyncExec(new Runnable() {
                public void run() {
                    try { redraw(); } catch(SWTException e) {}
                    // SWTException e' OK se la finestra principale
                    // viene chiusa dall'utente.
                }
            });
            TimeUnit.MILLISECONDS.sleep(pause);
        }
    } catch(InterruptedException e) {
        // Un modo accettabile per uscire
    } catch(SWTException e) {
        // Un modo accettabile per uscire: la finestra
        // principale viene chiusa dall'utente.
    }
}
}

public class ColorBoxes implements SWTApplication {
    private int grid = 12;
    private int pause = 50;
```



```
public void createContents(Composite parent) {  
    GridLayout gridLayout = new GridLayout(grid, true);  
    gridLayout.horizontalSpacing = 0;  
    gridLayout.verticalSpacing = 0;  
    parent.setLayout(gridLayout);  
    ExecutorService exec = new DaemonThreadPoolExecutor();  
    for(int i = 0; i < (grid * grid); i++) {  
        final CBox cb = new CBox(parent, pause);  
        cb.setLayoutData(new GridData(GridData.FILL_BOTH));  
        exec.execute(cb);  
    }  
}  
public static void main(String[] args) {  
    ColorBoxes boxes = new ColorBoxes();  
    if(args.length > 0)  
        boxes.grid = new Integer(args[0]);  
    if(args.length > 1)  
        boxes.pause = new Integer(args[1]);  
    SWTConsole.run(boxes, 500, 400);  
}  
} //:-~
```

Come nell'esempio precedente, la colorazione della finestra è gestita creando un **PaintListener**, che è dotato di un metodo **paintControl()**: questo metodo viene chiamato quando il thread SWT è pronto per disegnare il componente. **PaintListener** è registrato nel costruttore di **CBox**.

La differenza principale in questa versione di **CBox** è il metodo **run()**, che non può chiamare direttamente **redraw()** ma deve presentare **redraw()** al metodo **asyncExec()** sull'oggetto **Display**. Il metodo **asyncExec()** è simile a  **SwingUtilities.invokeLater()**: se sostituite questa operazione con una chiamata diretta a **redraw()**, il programma si interromperà.

Quando eseguite il programma, noterete alcune sottili righe orizzontali di disturbo attraversare una casella. Questo comportamento è dovuto al fatto che SWT in modalità predefinita non supporta il double buffering, a differenza di Swing. In effetti, eseguendo le due versioni SWT e Swing una di fianco all'altra, noterete che la versione Swing è più nitida. Tenete presente che è comunque possibile scrivere codice con supporto al double buffering anche in SWT: troverete dettagli ed esempi su [www.eclipse.org](http://www.eclipse.org).

**Esercizio 42** (4) Modificate `swt/ColorBoxes.java` in modo che inizi generando punti diffusi (“stelle”) sulla finestra; il programma dovrà poi modificare casualmente i colori di queste “stelle”.

### **SWT o Swing?**

È difficile tracciare un quadro chiaro con un'introduzione così breve, ma dovreste se non altro avere intuito che SWT, in molte situazioni, consente di adottare un approccio di programmazione più diretto rispetto a Swing. Tuttavia, la programmazione delle GUI in SWT può essere anche complessa e quindi la motivazione principale nella scelta di SWT dovrebbe essere quella di fornire all'utente una maggiore trasparenza nell'utilizzo della vostra applicazione, che avrà aspetto e comportamento molto simili a quelli delle applicazioni native. La seconda motivazione potrebbe riguardare l'eccellente prontezza di risposta fornita da SWT. In alternativa, anche Swing può comunque essere una scelta appropriata.

**Esercizio 43** (6) Scegliete uno qualsiasi degli esempi di Swing che non sia già stato convertito in questi ultimi paragrafi e trasformatelo in SWT. Questo è un ottimo esercizio per una classe di corso, poiché le soluzioni *non* sono presenti nella guida *The Thinking in Java Annotated Solution Guide*.

## **Riepilogo**

Le librerie GUI di Java hanno subito pesanti modifiche nel corso della vita di questo linguaggio. La libreria Java 1.0 AWT è stata criticata per la progettazione scadente; benché abbia permesso la creazione di programmi portabili, l'interfaccia grafica risultante era “ugualmente mediocre su tutte le piattaforme”. Era anche limitata, complessa e poco pratica da utilizzare, rispetto agli strumenti di sviluppo nativi disponibili per le varie piattaforme.

Quando Java 1.1 ha introdotto il nuovo modello a eventi e la tecnologia JavaBeans, le cose sono migliorate notevolmente: era finalmente possibile creare componenti GUI che potevano essere trascinati e incollati facilmente all'interno di un ambiente di programmazione visuale. Inoltre, la progettazione del modello a eventi e di JavaBeans evidenzia una grande considerazione per la facilità di programmazione e la manutenibilità del codice, argomenti per nulla evidenti nella libreria AWT 1.0. Ma la transizione si è completata soltanto con le classi Swing/JFC: grazie ai componenti Swing, la creazione di GUI multi-piattaforma può essere un'esperienza che non comporta troppi problemi.



Gli IDE rappresentano il vero motore della rivoluzione. Se state pensando di procurarvi un IDE commerciale per un linguaggio proprietario al fine di ottenere il meglio, dovrete sperare che il prodotto corrisponda alle vostre attese. Java è un ambiente aperto, che non solo permette la concorrenza tra gli ambienti IDE, ma la incorgaggia. E perché questi strumenti possano essere presi sul serio, devono supportare i JavaBean. Questo significa giocare su un terreno aperto: se si rende disponibile un IDE migliore, non sarete legati a quello che state utilizzando in quel momento, ma potrete utilizzare il nuovo strumento per incrementare il vostro rendimento. Questo tipo di competitività per gli ambienti IDE è del tutto inedito: il mercato che ne risulta può generare risultati molto positivi dal punto di vista del rendimento.

Questo capitolo ha voluto fornirvi un'introduzione alle potenzialità della programmazione visuale e gli strumenti indispensabili per affrontarla, così da poter valutare la semplicità di utilizzo di queste librerie. Ciò che avete visto finora probabilmente sarà sufficiente per la maggior parte delle vostre necessità quotidiane; tuttavia Swing, SWT e Flash/Flex possono offrirvi molto di più, in quanto strumenti di progettazione di interfacce grafiche estremamente completi: è quasi certo che esiste un metodo per realizzare qualsiasi cosa possiate immaginare.

### Risorse

Le presentazioni di Ben Galbraith, disponibili all'indirizzo [www.galbraiths.org/presentations](http://www.galbraiths.org/presentations), offrono un'eccellente introduzione a Swing e SWT.

*La soluzione degli esercizi è disponibile nel documento *The Thinking in Java Annotated Solution Guide*, in vendita all'indirizzo [www.mindview.net](http://www.mindview.net).*

# Appendice A

## Supplementi



All'integrazione di questo volume sono disponibili alcuni supplementi, tra cui la documentazione, i seminari e i servizi offerti sul sito web di MindView.

Questa appendice è la presentazione di tali supplementi, che vi consentirà di valutare se potranno esservi di aiuto.

Tenete presente che, anche se la maggior parte dei seminari è stata pensata per un pubblico vasto, potete richiedere corsi di addestramento privati e seminari aziendali presso la vostra sede.

### Supplementi scaricabili

Il codice di questo volume è scaricabile all'indirizzo [www.mindview.net](http://www.mindview.net). Si tratta fondamentalmente dei file di build Ant e di altri file di supporto necessari per costruire ed eseguire tutti gli esempi presentati nel libro.

Alcune parti del volume sono state inoltre convertite in formato elettronico, in particolare i seguenti argomenti:

1. clonazione di oggetti (*Cloning Objects*);
2. passaggio e restituzione di oggetti (*Passing & Returning Objects*);



3. analisi e progettazione (*Analysis and Design*);
4. porzioni di altri capitoli tratti da *Thinking in Java, terza edizione*, che l'autore non ha ritenuto appropriato includere nella versione cartacea della quarta edizione di questo testo.

## Thinking in C: i fondamenti di Java

Sul sito [www.mindview.net](http://www.mindview.net) potete scaricare gratuitamente il seminario *Thinking in C*: questa presentazione, creata da Chuck Allison e realizzata da MindView, è un corso multimediale in formato Flash che vi fornirà le nozioni introduttive alla sintassi, agli operatori e alle funzioni del linguaggio C, sui quali si basa la sintassi Java.

Tenete presente che l'utilizzo di questo seminario richiede l'installazione del software Flash Player di [www.macromedia.com](http://www.macromedia.com).

## Seminari Thinking in Java

La società dell'autore, MindView, Inc., eroga seminari e corsi di istruzione teorico-pratici di cinque giorni, aperti al pubblico e ai privati, in cui si sviluppano gli argomenti trattati in questo volume. Noto in precedenza come *Hands-On Java*, questo è il principale corso introduttivo che fornisce le nozioni necessarie per accedere agli altri seminari avanzati di MindView. Ogni lezione è realizzata con materiale selezionato da ogni capitolo ed è seguita da un periodo di esercitazioni controllate, in modo tale che ogni partecipante riceva la necessaria attenzione. Per informazioni su calendario, sedi dei corsi, testimonianze e ulteriori dettagli, consultate il sito [www.mindview.net](http://www.mindview.net).

## Seminario Hands-On Java su CD

Il CD *Hands-On Java* contiene una versione ampliata del seminario *Thinking in Java*, ed è anch'esso basato sui contenuti di questo libro; è stato concepito per fornire alcune delle esperienze del corso *live*, senza dovere sostenere l'onere del viaggio e del soggiorno. Il CD si compone di letture audio e dia-positive corrispondenti ai diversi capitoli del libro. L'autore ha realizzato personalmente il seminario: il materiale è in formato Flash, adatto quindi a qualunque piattaforma supporti il software Flash Player. Il CD *Hands-On Java* è in vendita sul sito [www.mindview.net](http://www.mindview.net), dove potrete scaricare anche versioni dimostrative di questo prodotto.



## Seminario Thinking in Objects

Questo corso introduce i concetti della programmazione a oggetti dal punto di vista del progettista, esplorando il processo di sviluppo e costruzione di un'applicazione, con particolare attenzione alle cosiddette "metodologie leggere" (*Agile Method* o *Lightweight Methodology*) e alla "programmazione estrema" (*XP, Extreme Programming*). Vi saranno illustrate le metodologie in generale, strumenti di utilità quali le tecniche di pianificazione *index-card* (descritte in *Planning Extreme Programming* di Beck e Fowler, Addison-Wesley, 2001), le schede CRC per la progettazione a oggetti, la programmazione "in coppie" (*pair programming*), l'*iteration planning*, i test unitari, il building automatizzato, il controllo del codice sorgente e argomenti analoghi. Il corso comprende un progetto XP, che verrà sviluppato durante la settimana formativa.

Se state per avviare un progetto e intendete usufruire di tecniche di progettazione orientate agli oggetti, potrete utilizzare le vostre tematiche come esempio e produrre una prima bozza di design nel corso della formazione.

Per informazioni su calendario, sedi dei corsi, testimonianze e ulteriori dettagli consultate il sito [www.mindview.net](http://www.mindview.net).

## Thinking in Enterprise Java

Questo libro ha avuto origine da alcuni dei capitoli più elaborati delle precedenti edizioni di *Thinking in Java*: non è una sorta di "secondo volume" di *Thinking in Java*, bensì un'attenta analisi degli argomenti più avanzati della programmazione d'impresa. Sebbene sia ancora in fase di sviluppo, attualmente è già disponibile per il download gratuito dal sito [www.mindview.net](http://www.mindview.net). Trattandosi di un libro separato, sarà possibile una sua successiva espansione per l'adeguata trattazione degli argomenti richiesti. Come per *Thinking in Java*, l'obiettivo è fornire un'introduzione agli elementi di base delle tecnologie di programmazione d'impresa, in modo che il lettore sia preparato ad affrontare in dettaglio questa materia. Tra gli argomenti citiamo i seguenti:

1. introduzione alla programmazione d'impresa;
2. programmazione di rete con socket e canali;
3. chiamate RMI (*Remote Method Invocation*);
4. connessione ai database;
5. servizi di rete (*Naming Service* e *Directory Service*);
6. servlet;



7. pagine JSP (*Java Server Page*);
8. tag, frammenti JSP e linguaggio di espressione;
9. automazione della creazione di interfacce utente;
10. enterprise JavaBeans;
11. XML;
12. servizi web;
13. test automatici.

Potete scaricare la versione corrente di *Thinking in Enterprise Java* all'indirizzo [www.mindview.net](http://www.mindview.net).

## Thinking in Patterns (with Java)

Uno dei passi fondamentali nella progettazione orientata agli oggetti è stato il cosiddetto "movimento dei design pattern", riassunto nel volume *Design Patterns*, di Gamma, Helm, Johnson e Vlissides (Addison-Wesley, 1995). Questo libro presenta 23 classi di problemi generici e le relative soluzioni, scritte soprattutto in C++, ed è un testo essenziale per un argomento che oggi è ormai quasi obbligatorio per qualunque programmatore OOP. *Thinking in Patterns* introduce i concetti di base di questi "modelli di progettazione", nonché alcuni esempi realizzati in Java: il libro non vuole essere una mera traduzione di *Design Patterns* in un diverso linguaggio, bensì una nuova prospettiva dall'ottica di Java; non si limita inoltre ai 23 modelli tradizionali, ma include altri concetti e tecniche per la soluzione dei problemi.

Il libro è nato come capitolo finale di *Thinking in Java, prima edizione*, ma assistendo al continuo sviluppo dei concetti è apparso chiaro che sarebbe dovuto diventare un volume autonomo. Al momento della stesura di queste note, il testo è ancora in fase di elaborazione: il materiale di base è stato più volte aggiornato con numerose presentazioni del seminario *Objects & Patterns*, ora ri-strutturato nei due corsi *Designing Objects & Systems* e *Thinking in Patterns*.

Troverete maggiori dettagli su questo libro all'indirizzo [www.mindview.net](http://www.mindview.net).

## Seminario Thinking in Patterns

Questo corso è un'evoluzione del precedente seminario *Objects & Patterns* tenuto da Bill Venners e dall'autore negli ultimi anni. Considerato il progressivo aumento del numero di argomenti, si è ritenuto opportuno suddividerlo in due parti: *Thinking in Patterns*, appunto, e *Designing Objects & Systems*.



Il seminario ricalca fedelmente il materiale e lo schema di presentazione del volume *Thinking in Patterns*, pertanto il modo migliore per conoscere i contenuti del corso è consultare i dettagli sul libro disponibili sul sito [www.mindview.net](http://www.mindview.net).

La presentazione è focalizzata soprattutto sul processo di evoluzione della progettazione, partendo da una soluzione iniziale e procedendo gradualmente nell'evoluzione logica e procedurale verso soluzioni progettuali più appropriate. L'ultimo progetto illustrato, la simulazione di un impianto per il riciclaggio dei rifiuti, si è evoluto nel corso degli anni: questo vi permette di considerare tale miglioramento come prototipo del modo in cui i vostri progetti, inizialmente semplici soluzioni a problemi specifici, potranno evolversi in un approccio più flessibile a un'intera categoria di problematiche.

Questo seminario vi aiuterà a:

1. potenziare in modo notevole la flessibilità dei vostri progetti;
2. implementare i concetti di estensibilità e di "riutilizzabilità" (*extensibility & reusability*);
3. incrementare il flusso di comunicazioni tra progetti, ricorrendo al linguaggio dei pattern.

Al termine di ogni incontro i partecipanti riceveranno un insieme di esercizi-modello da risolvere, nei quali saranno guidati alla realizzazione di codice che applica pattern specifici alla soluzione dei problemi di programmazione.

Per informazioni su calendario, sedi dei corsi, testimonianze e ulteriori dettagli, consultate il sito [www.mindview.net](http://www.mindview.net).

## Consulenza e revisione di progetti

MindView mette a disposizione servizi di consulenza, tutoraggio, revisione e applicazioni a supporto del ciclo di sviluppo dei vostri progetti, anche per il primo progetto Java della vostra azienda. Per conoscere disponibilità e dettagli dei servizi, consultate il sito [www.mindview.net](http://www.mindview.net).



# Appendice B

## Risorse



### Software

Il **JDK**, disponibile all'indirizzo <http://java.sun.com>. Anche se deciderete di utilizzare un ambiente di sviluppo di altri produttori, è sempre opportuno che abbiate a disposizione JDK, soprattutto nel caso vi troviate alle prese con un errore di compilazione. JDK è la pietra di paragone, il punto di riferimento in cui certamente troverete segnalato ogni possibile bug.

**Documentazione JDK** all'indirizzo <http://java.sun.com>, in formato HTML. È quasi impossibile trovare un testo sulle librerie standard di Java che non sia obsoleto o mancante di informazioni.

Malgrado la documentazione JDK di Sun presenti qualche bug e talvolta sia sintetica al punto da essere quasi inutilizzabile, ha il vantaggio di elencare *tutti* i metodi e le classi.

Alcuni utenti hanno inizialmente difficoltà a utilizzare una risorsa online invece di un testo cartaceo: è bene tuttavia che vi abituiate a servirvi dei documenti HTML, in modo da disporre della panoramica completa della situazione; in caso di difficoltà, potrete sempre ripiegare su un testo stampato.



## Editor e ambienti IDE

In questo settore la competizione è davvero agguerrita. Molte soluzioni sono gratuite e di solito quelle a pagamento offrono versioni dimostrative, pertanto non dovete far altro che provarle e verificare se qualcuna corrisponde alle vostre esigenze. Ecco alcune proposte:

**JEdit**, l'editor gratuito di Slava Pestov, è scritto in Java, quindi offre il vantaggio aggiuntivo di mostrarvi in azione un'applicazione desktop scritta in questo linguaggio. JEdit è supportato da un gran numero di plugin, molti dei quali scritti dalla comunità open source. Il software può essere scaricato all'indirizzo [www.jedit.org](http://www.jedit.org).

**NetBeans** è un ambiente IDE gratuito di Sun, disponibile all'indirizzo [www.netbeans.org](http://www.netbeans.org). Tra le funzionalità fornite, occorre segnalare la possibilità di "disegnare" grafiche mediante trascinamento, posizionamento e ridimensionamento, l'editing del codice e il debugging.

**Eclipse**, un progetto open source sostenuto, tra gli altri, da IBM. La piattaforma Eclipse è stata progettata con criteri di estensibilità, per consentirvi di produrre applicazioni standalone basate su questa piattaforma. È con questo software che è stata creata la libreria SWT (*Standard Widget Toolkit*) descritta nel Volume 3, Capitolo 2. Il software Eclipse può essere scaricato dal sito [www.eclipse.org](http://www.eclipse.org).

**IDEA di IntelliJ** è il software preferito di un gran numero di programmati Java, molti dei quali ritengono che IDEA sia sempre un passo avanti rispetto a Eclipse, forse per il fatto che IntelliJ non offre contemporaneamente un'interfaccia IDE e una piattaforma di sviluppo, ma si limita a fornire l'ambiente IDE. Potete scaricare una versione dimostrativa di IDEA dal sito [www.jetbrains.com](http://jetbrains.com).

## Libri

**Effective Java™ Programming Language Guide**, di Joshua Bloch (Addison-Wesley, 2001). Un testo che non deve mancare nella vostra libreria tecnica.

**Core Java™ 2, settima edizione, Volumi I e II**, di Horstmann & Cornell (Prentice Hall, 2005). Un testo ricco e completo, al quale molti ricorrono come risorsa di riferimento. L'autore raccomanda questo libro a chi desideri approfondire i concetti presentati in *Thinking in Java*.

**The Java™ Class Libraries: An Annotated Reference**, di Patrick Chan e Ronna Lee (Addison-Wesley, 1997). Benché obsoleto, questo libro contiene ciò che dovrebbe offrire JDK, vale a dire una descrizione sufficiente a rende-



re utilizzabile il software. Uno dei revisori di *Thinking in Java* ha affermato: "Se dovessi dare la preferenza a un solo titolo su Java sceglierrei questo". L'autore non è della stessa opinione, poiché ritiene che il volume sia troppo imponente (e costoso) e che la qualità degli esempi non sia soddisfacente. Rimane comunque uno dei testi da consultare a fronte di problemi, e in molti casi affronta gli argomenti in modo più approfondito di gran parte dei titoli alternativi. Rimane il fatto che *Core Java™ 2* esamina componenti di libreria molto più recenti.

***Java Network Programming, seconda edizione***, di Elliotte Rusty Harold (O'Reilly, 2000). L'autore ha dichiarato di avere sempre avuto difficoltà a comprendere l'argomento delle reti Java (delle reti in generale, a dire il vero), finché non ha scoperto questo titolo. Anche il sito web di supporto, Café au Lait ([www.cafeauait.org](http://www.cafeauait.org)), offre punti di vista stimolanti, opinioni notevoli e aggiornamenti accurati sullo sviluppo Java, non vincolati ad alcun produttore. La puntualità degli aggiornamenti vi permetterà di rimanere al passo con il mondo Java, sempre in movimento.

***Design Patterns***, di Gamma, Helm, Johnson e Vlissides (Addison-Wesley, 1995). Questo è il testo fondamentale che ha dato l'avvio al movimento della programmazione per modelli, di cui si parla diffusamente in *Thinking in Java*.

***Refactoring to Patterns***, di Joshua Kerievsky (Addison-Wesley, 2005). Una perfetta combinazione di refactoring e design pattern. L'aspetto più interessante di questo libro è che vi mostra come fare evolvere un progetto adattando i modelli alle vostre esigenze.

***The Art of UNIX Programming***, di Eric Raymond (Addison-Wesley, 2004). Malgrado Java sia un linguaggio multipiattaforma, la sua prevalenza su sistemi server ha dimostrato quanto sia importante la conoscenza dell'ambiente Unix/Linux. Questo testo è un'eccellente introduzione storica e filosofica a tale sistema operativo, una lettura avvincente anche per chi voglia soltanto conoscere alcuni aspetti dell'informatica.

## **Analisi e progettazione**

***Extreme Programming Explained, seconda edizione***, di Kent Beck e Cynthia Andres (Addison-Wesley, 2005). L'autore ha sempre sostenuto che dovrebbe esistere un modo migliore per affrontare lo sviluppo dei programmi, e XP sembra essere l'approccio giusto. L'unico libro che molti considerano di pari livello è *Peopleware*, descritto di seguito, che affronta principalmente l'ambiente e la cultura aziendale. *Extreme Programming Explained* tratta di programmazione e trasforma la maggior parte degli argomenti, anche le "scoperte" più recenti, in temi sorprendenti ed eccitanti. Gli autori di que-



sto libro arrivano addirittura a sostenere che le illustrazioni sono accettabili purché non si dedichi loro troppo tempo, tanto che sarebbero propensi a eliminarle (noterete infatti che il libro *non* ha il contrassegno di approvazione "UML stamp of approval").

Pensate come deve essere accettare di lavorare per una software house soltanto se questa adotta i principi della programmazione XP... Libri brevi, capitoli concisi e facili da leggere, idee entusiasmanti sulle quali riflettere. Iniziate a immaginare di lavorare in una simile atmosfera, che vi introdurrà in anteprima a un mondo completamente nuovo.

***UML Distilled, seconda edizione***, di Martin Fowler (Addison-Wesley, 2000). Avvicinandovi a UML potreste rimanerne intimiditi, a causa dei numerosi diagrammi e dettagli che caratterizzano questo linguaggio. Fowler ritiene che gran parte di questa sovrastruttura sia ridondante, pertanto la riduce all'essenziale. Per quasi tutti i progetti è sufficiente la padronanza di poche utility grafiche, e l'intento di Fowler è soprattutto quello di insegnarvi a realizzare un buon progetto senza soffermarsi eccessivamente sui dettagli: in effetti, di norma la lettura della sola prima parte del libro è più che adeguata. *UML Distilled* è un buon libro, non voluminoso, di cui apprezzerete la lettura: il primo che dovreste procurarvi per comprendere il linguaggio UML.

***Domain-Driven Design***, di Eric Evans (Addison-Wesley, 2004). Questo libro considera il modello di dominio (*domain model*) come strumento primario nel processo di progettazione. L'autore ritiene che questa sia una svolta determinante nella visione progettuale, in grado di portare i progettisti a conseguire un livello di astrazione ottimale.

***The Unified Software Development Process***, di Ivar Jacobson, Grady Booch e James Rumbaugh (Addison-Wesley, 1999). L'autore era prevenuto nei confronti di questo libro, che sembrava avere tutti i presupposti del tipico, noiosissimo testo universitario. In realtà si è rivelato una piacevole sorpresa, anche se in alcuni punti sembra che i concetti non siano molto chiari neppure agli autori. Nell'insieme, tuttavia, il libro non solo risulta scorrevole, ma anche ben scritto e, soprattutto, l'intero processo è illustrato in modo pratico e perfettamente coerente. Non si tratta di programmazione estrema (XP), perché non ne possiede la chiarezza in termini di test, ma è anch'essa parte dell'inesorabile valanga UML; una forza che, sebbene non avvicini direttamente gli utenti alla programmazione XP, ha fatto sì che molte persone, a prescindere dall'effettivo livello di esperienza con questa tecnologia, abbracciassero la "causa UML". L'autore ritiene che questo libro dovrebbe diventare il portabandiera del linguaggio UML, un testo che potrete leggere dopo aver affrontato *UML Distilled* di Fowler, qualora vogliate approfondire UML nei dettagli.

Prima di adottare una metodologia, è sempre utile conoscere il punto di vista di chi non si propone di vendervi nulla. È facile adottare una tecnologia senza comprendere appieno le aspettative che si hanno su di essa, o le caratteristiche che questa può offrire; per molti, è sufficiente il fatto che altri la utilizzino. Tuttavia alcuni hanno una curiosa perversione: se sono convinti che qualcosa risolverà i loro problemi, faranno di tutto per provarla (*la sperimentazione*, un approccio positivo); ma non appena si renderanno conto che i loro problemi non vengono risolti, raddoppieranno gli sforzi e annunceranno al mondo di avere scoperto qualcosa di eccezionale (questo significa *rinnegare*, un approccio tutt'altro che positivo). Molti ritengono che sia sufficiente fare salire sulla propria barca più gente possibile soltanto per non sentirsi soli, anche se la barca non ha una rotta precisa o se è destinata ad affondare.

Con ciò, l'autore non vuole affatto affermare che tutte le metodologie conducano al nulla: dovrete semplicemente armarvi fino ai denti di strumenti mentali che vi aiuteranno a rimanere su un livello di sperimentazione ("Non funziona, proviamo qualcos'altro"), ben lontani dalla condizione di "rifiuto" ("No, non è un problema: è tutto a posto, non c'è niente da cambiare"). L'autore ritiene che i titoli elencati di seguito, letti prima della fase di adozione di un metodo, possano fornirvi questi strumenti mentali.

**Software Creativity**, di Robert L. Glass (Prentice Hall, 1995). Secondo l'autore questo è il libro migliore che analizza le prospettive sull'intero problema delle metodologie: è una raccolta di saggi brevi e articoli che Glass ha scritto e in alcuni casi acquisito (P. J. Plauger è uno dei collaboratori), il risultato di molti anni di studio sull'argomento. Interventi piacevoli e lunghi a sufficienza da esprimere i concetti necessari: l'autore non vi annoierà né affligerà con vaneggiamenti di sorta. Né tantomeno tenterà di vendere fumo, dato che propone concetti suffragati da centinaia di riferimenti ad altri articoli e saggi. Tutti i programmati e i manager dovrebbero dedicarsi a questa lettura prima di guardare il pantano delle metodologie.

**Software Runaways: Monumental Software Disasters**, di Robert L. Glass (Prentice Hall, 1998). Il pregio di questo libro è che pone in risalto ciò di cui non si parla mai: la quantità di progetti che non soltanto falliscono, ma lo fanno in modo spettacolare. Molti pensano che a loro non potrà mai accadere nulla di simile, o quantomeno che non potrà più succedere, e questo è un atteggiamento sbagliato: se terrete sempre presente che le cose possono andare male, sarete sempre nella posizione giusta per poter migliorarle.

**Peopleware, seconda edizione**, di Tom DeMarco e Timothy Lister (Dorset House, 1999). Un titolo che tutti dovrebbero leggere: non soltanto è divertente, ma è un vero terremoto che metterà a soqquadro il vostro mondo e distruggerà



i vostri preconcetti. Sebbene DeMarco e Lister provengano da un'esperienza di sviluppo software, il libro prende in esame i progetti e i team operativi in generale, ma mettendo a fuoco le *persone* e le loro necessità, non tanto la tecnologia e le sue esigenze. In *Peopleware* vedrete come creare un ambiente in cui le persone siano soddisfatte e produttive, senza il solito elenco di regole a cui occorre conformarsi per diventare perfetti "componenti" di una macchina: un'attitudine, quest'ultima, che spesso è causa di un falso atteggiamento da parte di molti programmati, i quali aderiscono con finto entusiasmo a ogni nuovo metodo X, e poi tornano a fare esattamente come prima.

*Secrets of Consulting: A Guide to Giving & Getting Advice Successfully*, di Gerald M. Weinberg (Dorset House, 1985). Un libro superbo, uno dei preferiti dell'autore, perfetto per chi si avvicina alla professione di consulente o per chi, come consulente, voglia progredire nella professione. Capitoli brevi, integrati con storie e aneddoti che illustrano come arrivare al nocciolo del problema con il minimo sforzo. Altri testi interessanti sull'argomento sono *More Secrets of Consulting*, pubblicato nel 2002, e molti altri lavori di Weinberg.

*Complexity*, di M. Mitchell Waldrop (Simon & Schuster, 1992). Questo libro è la cronaca dell'incontro che un gruppo di scienziati di varie discipline ha tenuto a Santa Fé, New Mexico, allo scopo di discutere problemi reali che le singole discipline non sono in grado di risolvere: il mercato azionario in economia, l'origine della vita in biologia, le motivazioni comportamentali in sociologia ecc. Dalla combinazione di fisica, economia, chimica, matematica, informatica, sociologia e altre scienze si sta sviluppando un approccio multidisciplinare ai problemi, ma soprattutto sta emergendo una *forma mentis* alternativa nei confronti di queste tematiche ultra-complesse: tale approccio è ben distante dal determinismo matematico e dall'illusione che un'equazione basti a predire qualsiasi comportamento, ma è volto all'osservazione preliminare, alla ricerca di un modello e al tentativo di emularlo a tutti i costi (nel libro è descritto, per esempio, l'emergere degli algoritmi genetici). Secondo l'autore, questo è un approccio prezioso per affrontare la gestione di progetti software sempre più complessi.

## Python

*Learning Python, seconda edizione*, di Mark Lutz e David Ascher (O'Reilly, 2003). Una valida introduzione al linguaggio preferito dell'autore, oltre che un eccellente compagno per Java. Il libro include un'introduzione a Jython, che vi consente di combinare Java e Python in un unico programma: l'interprete Jython è compilato in puro bytecode Java, quindi non vi serve niente di speciale nel caso dobbiate utilizzarlo. Un'unione di due linguaggi che preannuncia grandi possibilità.



## Bibliografia dell'autore

Nessuno di questi testi è più a catalogo, ma potreste trovarne alcuni nelle librerie specializzate.

**Computer Interfacing with Pascal & C** (pubblicato in proprio per i tipi di Eisy, 1988. Disponibile in vendita soltanto presso [www.mindview.net](http://www.mindview.net)). Un'introduzione all'elettronica, che risale ai tempi in cui CP/M era ancora re e DOS muoveva i primi passi: per pilotare vari progetti elettronici, l'autore si è servito di linguaggi di alto livello e spesso della porta parallela del computer. Il libro è un adattamento degli articoli scritti dall'autore per la prima (e migliore) rivista con cui ha collaborato, Micro Cornucopia, che purtroppo ha cessato le pubblicazioni molto tempo prima che Internet diventasse popolare. La redazione di questo libro è stata una delle esperienze editoriali più soddisfacenti per l'autore.

**Using C++** (Osborne/McGraw-Hill, 1989). Uno dei primi libri su C++, ormai fuori catalogo e sostituito dalla seconda edizione, intitolata *C++ Inside & Out*.

**C++ Inside & Out** (Osborne/McGraw-Hill, 1993). Si tratta della seconda edizione di *Using C++*. Il codice C++ esaminato in questo libro è ragionevolmente accurato, ma risale al 1992 circa, e per questo motivo è stato sostituito da *Thinking in C++*. Troverete maggiori dettagli sul sito [www.mindview.net](http://www.mindview.net), dove potrete anche scaricare il codice sorgente.

**Thinking in C++, prima edizione** (Prentice Hall, 1995). Questo libro ha vinto il premio *Software Development Magazine Jolt Award* quale miglior libro dell'anno.

**Thinking in C++, seconda edizione, Volume 1** (Prentice Hall, 2000). Scaricabile da [www.mindview.net](http://www.mindview.net) e aggiornato secondo gli standard finalizzati del linguaggio.

**Thinking in C++, seconda edizione, Volume 2**, scritto in collaborazione con Chuck Allison (Prentice Hall, 2003): il libro è scaricabile dal sito [www.mindview.net](http://www.mindview.net).

**Black Belt C++: The Master's Collection**, a cura di Bruce Eckel (M&T Books, 1994). Questo libro, fuori catalogo, è una raccolta di testi di varie "autorità" in materia di C++, basati sulle loro presentazioni alla conferenza Software Development Conference, presieduta dall'autore. È stata la copertina di questo libro a stimolare Eckel nella progettazione di tutte le successive copertine dei suoi lavori.

**Thinking in Java, prima edizione** (Prentice Hall, 1998). La prima edizione di questo libro ha vinto i premi *Software Development Magazine Productivity*



*Award, Java Developer's Journal Editor's Choice Award e JavaWorld Reader's Choice Award* come miglior libro. Questa edizione può essere scaricata dal sito [www.mindview.net](http://www.mindview.net).

***Thinking in Java, seconda edizione*** (Prentice Hall, 2000). Questa edizione ha vinto il premio *JavaWorld Editor's Choice Award* quale miglior libro, ed è scaricabile da [www.mindview.net](http://www.mindview.net).

***Thinking in Java, terza edizione*** (Prentice Hall, 2003). Questa edizione ha vinto il premio *Software Development Magazine Jolt Award* quale miglior libro dell'anno e numerosi altri premi, elencati in quarta di copertina. Il testo è scaricabile dal sito [www.mindview.net](http://www.mindview.net).

# Indice analitico

## A

ActionScript e MXML 348  
active objects (oggetti attivi) 210  
adattatori 246  
Adobe Flash 343  
  costruzione di client web in,  
    con Flex 344-359  
applet 222  
appunti 301  
aree di testo 232-234  
Atomic, classe 203  
atomicità 57-65  
AWT (*Abstract Windowing Toolkit*) 219

## B

barre di avanzamento 296-298  
bean  
  complessi 331-335  
  pacchettizzazione dei 340-342  
supporto avanzato ai 342-343  
BeanInfo, estrazione di,  
  con Introspector 324-331  
bloccaggio 4  
blocco sincronizzato 68  
BorderLayout 234-236  
bordi 260-261  
BoxLayout 238  
busy wait 99, 104

## C

callback 229  
cambio di contesto (*context switch*) 4  
campi di testo 257-260  
caselle  
  combinare 266-267  
  di riepilogo 268-270  
  di scelta 263-264  
classi atomiche 65-67  
code 118-124  
  di eventi (*event queue*) 224  
  sincronizzate (*synchronized queue*) 118  
codice  
  migliorare la progettazione del 7-8  
  variazioni sul 30-37  
Command, design pattern 14  
componenti di libreria 134-161  
componenti Swing 251-301  
  barre di avanzamento 296-298  
  bordi 260-261  
  campi di testo 257-260  
  caselle combinate 266-267  
  caselle di riepilogo 268-270  
  caselle di scelta 263-264  
  controllo della disposizione dei  
    234-239  
  cursori scorrevoli 296-298  
  disegno 283-288  
  elenchi a discesa 266-267  
  finestre di dialogo 288-295



finestre di messaggio 271-274  
HTML nei 295-296  
icone 255-257  
menu 274-281  
menu pop-up 282-283  
mini-editor 261-262  
pannelli a schede 270-271  
pulsanti 251-252  
pulsanti radio 264-266  
tooltip 257  
comportamento auto-escludente (o esclusivo) 253  
concorrenza 1-218  
Condition, oggetto, utilizzo di 114-117  
contenitori 349-351  
non soggetti a lock 193-203  
privi di lock (*lock-free*) 194  
controlli 349-351  
corsa critica (*race condition*) 49  
CountDownLatch, oggetto 134-137  
CSP (*Communicating Sequential Processes*) 214  
CSS (*Cascading Style Sheets*) 352  
cursori scorrevoli 296-298  
CyclicBarrier, classe 137-141

## D

data binding 356  
deadlock 127-134  
DelayQueue, classe 141-144  
driver (*codec*) 351  
DTO (*Data Transfer Object*) 357

## E

eccezioni, intercettazione delle 43-47  
effetti 351-352  
elenchi a discesa 266-267  
esclusione reciproca (*mutex*) 53  
esclusione reciproca (*mutual exclusion*) 51  
esecutore di thread singolo (*single thread executor*) 213

EvenGenerator, sincronizzazione di 53-54  
eventi  
cattura degli 228-232  
multipli, monitoraggio di 247-251  
tipi di 240-247  
Exchanger, classe 158-161  
Executor, utilizzo di 13-17

## F

finestre di dialogo 288-295  
per i file 292-295  
finestre di messaggio 271-274  
Flex  
collegamento ai dati di 356-357  
compilazione e distribuzione 357-359  
connessione a Java con 353-356  
contenitori e controlli 349-351  
effetti e stili 351-352  
eventi 352-353  
modelli dei dati 356-357  
riquadri (*tile*) 349  
FlowLayout 236  
flushing 59  
forking 6  
framework di visualizzazione 226-227

## G

gestore di layout (*layout manager*) 228  
giardino ornamentale 78-83  
GIF, immagine 255  
GridBagLayout 237-238  
GridLayout 237  
GUI (*Graphical User Interface*) 219

## H

hand-over-hand locking 57  
HTML 295-296

**I**

icone 255-257  
**IDE (Integrated Development Environment)** 321  
 implementazioni Map, confronto 201-203  
 Index, funzionalità 234  
 interfacce grafiche (GUI) 219-380  
 interfacce utente  
   reattive, creazione di 40-42  
 interrupt, controllo di 95-98  
 interruzione non brusca (*graceful shutdown*) 49  
 Introspector, classe 325-332

**J**

JAR, file 340  
**JavaBeans**  
   definizione di 322-324  
   e programmazione visuale 320-343  
   sincronizzazione 335-340  
   ulteriori risorse su 343  
 Java web Start 301-307  
**JButton**, costruttore 227-228  
**JDialog**, classe 288  
**JDK** 220  
**JFC (Java Foundation Classes)** 220  
**JNLP (Java Network Launch Protocol)** 301-307  
**JOptionPane.showConfirmDialog()**, metodo 271  
**JOptionPane.showMessageDialog()**, metodo 271

**L**

linguaggi funzionali 6  
 linguetta (*tab*) 270  
 listener (ascoltatori) 239  
   semplificazione mediante gli adattatori di 246-247  
   tipi di 240-247

**Lock, oggetto**

  contenitori non soggetti a 193-203  
   di tipo ReadWriteLock 206-209  
   ottimistico 203-206  
   utilizzo di 114-117

**M**

menu 274-281  
   pop-up 282-283  
 microbenchmarking 184  
**Microsoft Visual BASIC (VB)** 321  
**mini-editor** 261-262  
 modello di memoria 60  
 moduli (*form*) 223  
 multicast 334  
 multicore, processori 3, 58  
 multitasking 5  
 multithreading cooperativo 7  
 mutex, confronto tra tecnologie 182-193  
**MVC (Model-View-Controller)** 357  
**MXML**  
   compilare in 346-348  
   e ActionScript 348

**N**

notifyAll(), metodo 99-106  
   confronto di, con notify() 106-109

**O**

oggetti  
   attivi 210-215  
   disconnessi (*disconnected object*) 213  
   Lock esplicativi, utilizzo di 54-57  
   ottimizzazione, problemi di 194-201

**P**

paintComponent(), metodo 284  
 pannelli a schede 270



- pianificatore (*task scheduler*) 54  
pipe, utilizzo delle per l'I/O tra task 124-127  
pluggable look & feel, approccio 298  
pool (*object pool*) 15, 154  
posizionamento assoluto 238  
preempted 42  
prestazioni, ottimizzazione delle 182-209  
PriorityBlockingQueue, classe 145-148  
processi 5  
programmazione  
  a eventi (*event-driven*) 4, 228  
  concorrente 3-8  
  sequenziale 1  
  terminologia 37-38  
  visuale 321  
programmi, velocità di esecuzione dei 3-7  
proprietà  
  collegate (*bound*) 343  
  indicizzate (*indexed property*) 343  
  vincolate (*constrained*) 343  
pulsanti 251-252  
  costruzione di 227-228  
  gruppi di 253-254  
pulsanti radio 264-266  
punto di entrata (*entry point*) 306  
punto morto (*deadlock*) 127  
punto terminale (*endpoint*) 353
- R**
- riflessione 242, 325  
risorse  
  accesso improprio alle 47-50  
  condivisione delle 46-78  
  risoluzione di conflitti tra 50-57  
RMI (*Remote Method Invocation*) 325  
Runnable JPanel, classe 318
- S**
- sandbox 222, 301  
ScheduledExecutor, classe 148-153
- scorrimento (scrolling) 233  
semafori 153-158  
  contatore (*counting semaphore*) 153  
servlet 2  
setEditable() 266  
setPriority(), metodo 21-24  
sezioni critiche 67-74  
shutdown 14  
simulazione 161-182  
sincronizzazione 74-76  
sistemi di messaggistica 8  
sleep(), metodo 19-21  
sottotask 8  
stato di interrupt (*interrupted status*) 95  
stato inconsistente (*damaged*) 84  
stenografia dichiarativa (*declarative shorthand*) 348  
Strategy, design pattern 144  
struts and glue, approccio 238  
strutture ad albero 301  
SVG (*Scalable Vector Graphics*) 351  
Swing 2, 220  
  alternative a 343-344  
  concorrenza 307-320  
  modello a eventi di 239-250  
  modifica dell'aspetto dell'interfaccia 298-301  
  nozioni di base su 223-227  
  selezione di componenti 251-301  
SWT (*Standard Widget Toolkit*) 344  
concorrenza in 376-379  
creazione di applicazioni 360-380  
eliminazione del codice ridondante 365-367  
grafica 373-376  
installazione di 360-361  
menu 367-368  
pannelli a schede 369-373  
pulsanti ed eventi 369-373
- T**
- tabelle 301  
task  
  bloccati, terminare i 83-84



- blocco di, causato da mutex 92-94
  - chiusura dei 78-98
  - cooperazione tra 98-127
  - definizione dei 9-11
  - di lunga durata 308-317
  - interruzione dei 84-94
  - messaggio in pausa dei, con sleep() 19-21
  - ottenere valori di ritorno dai 17-19
  - utilizzo delle pipe per l'I/O tra 124-127
  - Template Method, design pattern 191
  - tempo di attivazione (*trigger time*) 144
  - thread
    - collegamento dei 38-40
    - d'invio degli eventi (*event dispatch thread*) 224
    - demoni (*daemon*) 24-30
    - di esecuzione (*thread of execution*) 8
    - di tipo event-dispatching 16
    - dormienti (*sleep*) 40
    - funzionalità della libreria per la sicurezza dei 137
    - gestione dei 8-47
    - gruppi di 42
    - memoria locale di 76-78
    - priorità 21-24
    - singolo (*single-threaded*) 46
    - stati dei 83
  - Thread, classe 11-13
  - Thread, oggetto 307
  - threading 6
  - preemptive 7
  - visuale 318-320
  - thread scheduler 10, 12
  - tooltip 257
- U**
- unicast 334
- V**
- velocità di esecuzione 3-7
  - volatilità 57-65
- W**
- wait(), metodo 99-106
  - widget 349, 369
  - word tearing 58
- Y**
- yield(), metodo 23-24
  - yielding 23-24

