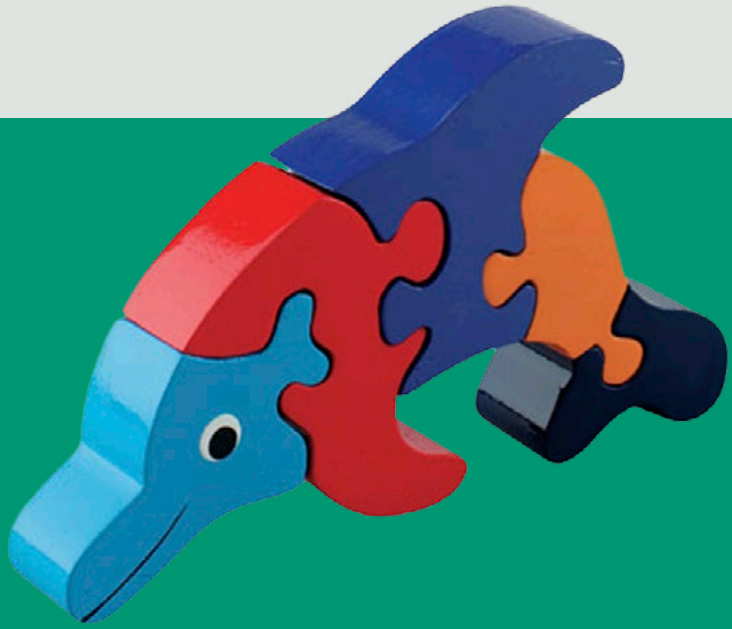
MBook

Luca Vetti Tagliati

verso Java 8

Appunti per lo sviluppatore in Java SE 7



mokabyte

Luca Vetti Tagliati

Verso Java 8
Appunti per lo sviluppatore in Java SE 7

versione 1
febbraio 2013

MokaByte
www.mokabyte.it

Sommario

Capitolo 3. Fork et impera

Capitolo 4. NIO 2

Capitolo 5. Le feature restanti

Capitolo 6. Aspettando la release 8

Appendice A. Multi-Threading in Java

Appendice B. New Input-Output API

Appendice C. Visitor pattern

Appendice D. Stile funzionale in un chicco di grano

Riferimenti bibliografici

Introduzione

Alla mia famiglia
"Papà ora 'scende' dal computer. Andiamo a giocare!".

Obiettivi Lo scopo di questo ebook è di fornire una descrizione dettagliata e operativa dell'ultima versione di Java: **Java SE 7**, nome in codice **Dolphin** ("delfino"), rilasciata ufficialmente da Oracle il 7 luglio del 2011 (l'apertura al pubblico è stata posticipata di 21 giorni). Le nuove feature sono illustrate in modo operativo, spesso mostrando porzioni del codice sorgente e diagrammi UML, principalmente per enfatizzarne la struttura. Sebbene Java SE7 sia disponibile da più di un anno e mezzo, i dati di mercato mostrano che il suo utilizzo è ancora marginale. La speranza è dunque che questo libro possa risultare un valido ausilio al personale di lingua italiana coinvolto nella migrazione dei vari sistemi alla versione Java SE 7. A tal fine, il motivo conduttore seguito per la redazione del libro è stato quello di fornire una guida pratica e immediata per gli sviluppatori. Ecco dunque spiegato il titolo: è un "quaderno degli appunti" da consultare per lo svolgimento del lavoro quotidiano. Grande enfasi è stata attribuita alle sperimentazioni in codice, attualizzando gli insegnamenti dei nostri saggi avi: *verba volant... codex manet*.

A chi è rivolto Questo libro è dedicato alla comunità di programmatori Java. In particolare, vuole essere un valido supporto a tutti coloro che si accingono a migrare sistemi in Java SE 7 o per chi semplicemente è interessato ad approfondire le proprie conoscenze relative a questa versione, apprendendo anche dettagli che talune volte sfuggono alla letteratura ufficiale. Come è lecito attendersi, la conoscenza di Java è un prerequisito.

Genesi Questo ebook è l'evoluzione della serie di articoli pubblicati dall'autore, a partire da marzo 2012 su **www.MokaByte.it** sul tema di Java e della sua evoluzione (cfr. in "Riferimenti"). Visto l'interesse riscontrato, si è deciso di trasformare questa esperienza in un libriccino elettronico che potesse soddisfare le esigenze del pubblico di sviluppatori Java che necessita di una guida veloce e pratica. Non si tratta però della pedissequa trascrizione degli articoli: il libro è **organizzato in maniera diversa** ed è frutto di una **rielaborazione attualizzata** di quella serie.

Struttura Il libro è organizzato nei seguenti capitoli:

Introduzione Giunti a questo punto, avete già letto una parte di questo capitolo dedicato ad argomenti quali: obiettivi, il potenziale pubblico dei lettori, la struttura e gli immancabili ringraziamenti.

Capitolo 1. Java SE7 in breve In questo capitolo forniamo una breve storia della versione Java SE 7 corredata da un sintetico resoconto degli aggiornamenti apportati, ma senza entrare troppo nei particolari: per questo scopo sono presenti i rimanenti capitoli.

Capitolo 2. La "moneta" dei cambiamenti Questo capitolo è dedicato agli aggiornamenti apportati alla sintassi e semantica del linguaggio di programmazione Java (e non alle sue librerie) attraverso il progetto **Coin** ("moneta"). In particolare, sono state introdotte le seguenti variazioni: costrutti **switch** con stringhe, estensione del costrutto **catch** per gestire eccezioni multiple, **try-with-resource**, eccezioni con controllo di tipo più preciso, inferenza di tipo, miglioramento dei **warnings**, introduzione dei tipi letterali binari e possibilità di utilizzare degli **underscore** nei letterali numerici.

Capitolo 3. Fork et impera In questo capitolo presentiamo ulteriori miglioramenti introdotti nel package della concorrenza (**java.util.concurrent**). In particolare, viene presentato il framework del **Fork and Join**, la nuova funzione random, la nuova coda bloccante e il **phaser**.

Capitolo 4. NIO2 In questo capitolo sono presentate le modifiche relative all'API NIO (New Input-Output): si tratta di una serie di variazioni significative che hanno portato alla formulazione della versione **NIO2**. Le principali innovazioni sono condensate nei nuovi package: **java.nio.file**, **java.nio.file.attribute**, **java.nio.file.spi** e **com.sun.nio.sctp**. Obiettivi principali di questi cambiamenti sono: aumento dell'indipendenza dalla piattaforma di esecuzione, implementazione di ulteriori servizi di alto livello, fornitura di servizi per lavorare con i metadati e i collegamenti simbolici, implementazione del supporto del protocollo SCTP.

Capitolo 5. Le restanti feature Questo capitolo si occupa degli aggiornamenti a corallario della versione Java SE 7, come per esempio il passaggio all'**Unicode 6.0.0**, il supporto estensibile per i codici standard delle valute, le variazioni alle impostazioni della localizzazione (**Local**), le nuove feature della GUI (**Nimbus**, **JLayer**, **Translucent window**), e così via.

Capitolo 6. Aspettando la release 8 In questo capitolo facciamo un'incursione nella versione **Java SE 8**. In particolare, presentiamo lo stato dei lavori del progetto **Jigsaw** ("puzzle"), sebbene sia notizia di autunno 2012 che esso verrà verosimilmente posticipato a Java 9. Vediamo poi alcuni elementi del progetto **Coin** tralasciati da Java SE 7 che potrebbero confluire in Java SE 8 e poi presentiamo quella che può essere considerata la più attesa feature di Java SE 8: le **Lambda Expressions**. Si tratta di una piccola rivoluzione, giacché l'introduzione dello stile funzionale richiede di modificare la sintassi e la semantica di Java, l'ambiente JDK e l'intero set delle librerie: un po' come avvenuto con

l'introduzione dei **Generics**, ma con una mutazione molto più profonda. Come vedremo, l'implementazione di questa feature sta lasciando parte della comunità Java insoddisfatta per via del ridotto supporto allo stile funzionale. Tuttavia bisogna tenere a mente che l'evoluzione di Java è un esercizio per funamboli dati i tanti vincoli a cui è necessario sottostare: compatibilità a ritroso in primis.

Appendice A. Multi-threading in Java Abbiamo concentrato nelle appendici alcuni temi importanti per una visione di insieme ma che rischiavano di risultare ridondanti negli altri capitoli. In particolare, l'Appendice A si occupa del multi-threading in Java, anche da un punto di vista "storico", perché ci permette di comprendere al meglio alcuni meccanismi.

Appendice B. New Input-Output API Nell'Appendice B, ci occupiamo della API NIO, la cui conoscenza è propedeutica alla comprensione della nuova NIO2 di cui ci occupiamo al Capitolo 4.

Appendice C. Visitor pattern Questa appendice è dedicata alla presentazione del "pattern del visitatore", nell'ottica di fornire informazioni di background per la piena comprensione della feature WalkFileTree, introdotta con Java SE 7.

Appendice D: Stile funzionale in un chicco di grano Nell'ultima appendice presentiamo la programmazione funzionale, un importante paradigma che ha ispirato diversi linguaggi di programmazione, basti citare LISP e Scala. Lo scopo è permettere una migliore comprensione del progetto Lambda, delle relative sfide e del suo impatto sulla piattaforma Java.

Riferimenti bibliografici Non manca un breve ma significativo elenco delle opere di riferimento citate nel volumetto, che possono rappresentare lo spunto per i lettori che intendano approfondire i diversi argomenti.

Ricorso a pratiche sconsigliate... Molti degli esempi mostrati includono alcune pratiche assolutamente "deplorevoli" i cui esempi classici sono:

- l'utilizzo del metodo **main** per eseguire una sorta di test della classe;
- il massiccio ricorso al **System.out** per produrre l'output su console.

Si tratta di pratiche assolutamente sconsigliate [Vetti Tagliati, 2008]. In particolare, se uno sviluppatore fosse tentato di ricorrere al metodo **main** per eseguire il test di una classe, dovrà immediatamente cambiare idea e creare un apposita classe di test. Questo perché tali classi di test sfruttano appositi framework di test come **JUnit**, e forniscono un corredo indispensabile al codice. Allo stesso modo, non si dovrebbe utilizzare **System.out** ma favorire l'utilizzo di un apposito log.

Detto ciò, alcune pratiche deprecabili sono state utilizzate in questo contesto esclusivamente per questioni di impaginazione e per evitare di dover riportare

lunghi codici che, verosimilmente, finirebbero per distrarre il lettore dall'argomento centrale del libro: la piena comprensione delle nuove feature di Java SE7.

UML Una delle peculiarità offerte da questo libriccino è il ricorso ai diagrammi UML, principalmente alla notazione dei digrammi delle classi al fine di raggiungere un duplice obiettivo: documentare le nuove feature introdotte con Java SE7 attraverso diagrammi che valgono intere pagine di testo, e fornire al lettore esempi pratici di utilizzo della notazione UML per disegnare e documentare il codice Java.

Breve biografia dell'autore Luca Vetti Tagliati ha conseguito il PhD presso il Birkbeck College (University of London) e, negli anni, ha applicato la progettazione/programmazione OO, Component Based e SOA in diversi settori del trading bancario, prevalentemente in ambienti Java SE/EE. Dopo aver lavorato a lungo nella City di Londra per le più importanti banche di investimento (Goldman Sachs, Lehman, Deutsche Bank, UBS, HSBC), attualmente ricopre l'incarico di Senior Lead Global Architect per un importante gruppo bancario in Svizzera. Ha collaborato attivamente con John Daniels (coautore del libro "UML components") e Frank Armour (coautore del libro "Advanced Use Case Modeling"). È autore del libro "UML e ingegneria del software", pubblicato nel 2003 da HOPS/Tecniche Nuove, di "Java Best Practice: i migliori consigli per scrivere codice di qualità", pubblicato nel 2008 dal medesimo editore, nonché, ovviamente, di questo... È stato invitato a Los Angeles e Boston per presentare i suoi paper in convegni inerenti Software Engineering & Architecture. Nel tempo libero (!) continua a ostinarsi nell'applicare i pattern al gioco del calcio... nonostante gli svariati suggerimenti, provenienti da più parti, a lasciar stare.

Ringraziamenti Ed eccoci qua alla consueta sezione dedicata ai ringraziamenti. In primis, come buona tradizione italica insegna, ringrazio mia moglie Vera, che ben presto verrà santificata come prima donna martire di Duke, e mio figlio Francesco Alexander che ha l'immenso merito di esistere. La mia personale riconoscenza va poi agli altri due membri del team: Francesco Saliola e Giovanni Puliti che, affetti da conclamato squilibrio, hanno il pregio o difetto (a voi l'ardua sentenza) di continuare teneramente a seguirmi in queste mie imprese: ormai ho perso ogni speranza di seminarli!

Capitolo 1
Java SE 7 in breve

Introduzione

In questo capitolo forniamo alcune informazioni "storiche" su Java SE7 ed una rapida overview delle principali feature introdotte, senza però entrare nei dettagli: questo è l'obiettivo dei successivi capitoli.

Il nome in codice di Java SE7 è Dolphin ("delfino") e il **rilascio ufficiale è avvenuto il 7 luglio del 2011** mentre per l'apertura al pubblico si è dovuto attendere per tre settimane. Il progetto è stato organizzato in 13 milestone: il "feature complete" è stato raggiunto il 23 dicembre 2011, mentre la preview per gli sviluppatori è iniziata solo il 17 febbraio 2012, quindi un paio di mesi fa(cfr. [JDK7, 2011]).

Si tratta di una versione molto importante soprattutto per le implicazioni, diciamo così, "politiche" ed "economiche". Come confermato dallo stesso Gosling, "Java SE 7 è importante non tanto per una caratteristica particolare, ma per il fatto che Oracle sia riuscita a rompere l'impasse politico nel JCP che l'aveva rinviata per molto tempo" ("Java SE 7 is important not for any particular feature but for the fact that Oracle was able to bust the political logjam in the JCP that had delayed it for so very long." [Krill, 2011]).

OpenJDK

Nel maggio del 2007, la fu Sun Microsystem rilascia finalmente il kit OpenJDK, soddisfacendo una continua pressione da parte della Java Community. Si tratta di un'implementazione aperta e gratuita del linguaggio di programmazione Java. La licenza prescelta è GNU General Public License (GPL), con un'eccezione di collegamento (linking exception) che esenta Java Class Library dai termini della licenza GPL [OpenJDK].

E venne Oracle

Nel gennaio del 2010, Sun Microsystem viene acquisita da Oracle Corporation con un accordo da 7.4 miliardi di dollari. Il 2 aprile del 2010 James Gosling, da molti considerato il "padre di Java", dopo aver severamente criticato la gestione

dell'acquisizione da parte di Oracle (soprattutto per questioni contrattuali) si dimette. Come lui stesso asserisce nel suo blog [Gosling, 2010]: "Sì, infatti, le voci sono vere: mi sono dimesso da Oracle una settimana fa (2 aprile). [...] Quasi tutto quello che potrei dire che sia accurato e veritiero farebbe più male che bene. La parte più difficile è non essere più parte del team formato da tutte le grandi persone con cui ho avuto il privilegio di lavorare nel corso degli anni". ("Yes, indeed, the rumors are true: I resigned from Oracle a week ago (April 2nd). I apologize to everyone in St Petersburg who came to TechDays on Thursday expecting to hear from me. I really hated not being there. As to why I left, it's difficult to answer: Just about anything I could say that would be accurate and honest would do more harm than good. The hardest part is no longer being with all the great people I've had the privilege to work with over the years.").

L'azienda a cui approda è Google alla quale tuttavia non resta a lungo; e infatti dopo cinque mesi decide di dimettersi per iniziare una nuova avventura: inizia a collaborare con Liquid Robotics (cfr. [Taft, 2011]), una start-up che si occupa di acquisizione ed elaborazione di dati relativi ai fondali oceanici. Come descritto dallo stesso James Gosling "Liquid Robotics affronta un problema estremamente complesso che fa del bene al mondo ed è incredibilmente 'fico': Liquid Robotics può cambiare completamente il modo in cui guardiamo agli oceani. Saremo in grado di ottenere una vasta gamma di dati dettagliati più a buon mercato e più pervasiva rispetto a qualsiasi altro metodo. Si tratta di problemi legati alle grandi dimensioni di dati e di controllo: entrambi sono affascinanti per me e sono state miei passioni per anni." ("Liquid Robotics tackles a rocket science problem that does good for the world and is incurably cool, Liquid Robotics can totally change the way we look at oceans. We'll be able to get a wide variety of detailed data more cheaply and pervasively than any other way. It involves a large data problem and a large-scale control problem, both of which are fascinating to me and have been passions of mine for years.").

Il famoso piano B

La release Java SE 7 era partita con un piano molto ambizioso che tra l'altro includeva l'introduzione delle espressioni Lambda e la modularizzazione del JDK (progetto Jigsaw, *puzzle*). Con queste premesse si trattava di una major release decisamente rivoluzionaria. Tuttavia, il piano non andò esattamente come programmato, scenario peraltro tipico dei progetti software di grandi dimensioni. A complicare le cose intervenne l'acquisizione di Sun Microsystems da parte di Oracle, con l'inevitabile avvicendamento alle sedie di comando, che finì per rendere la situazione ancora più complicata.

Già nel settembre del 2010 Mark Reinhold, Sun e Oracle Chief Architect per la piattaforma Java e OpenJDK, scriveva nel suo blog un post dal titolo "There's not a moment to lose" (cfr. [Reinhold, 2010]) ("Non c'è un attimo da perdere"), con un'introduzione piuttosto eloquente: "È divenuto chiaro da tempo che il recente programma di sviluppo JDK 7 è, a dir poco, irrealistico. Abbiamo creato quel piano oltre nove mesi fa, prima dell'acquisizione di Sun da parte di Oracle. Il processo di post-acquisizione e di integrazione, purtroppo, ha richiesto più tempo di quello immaginato da tutti noi, ma ora siamo pronti e in grado di concentrarci su questa versione con un team più grande (e ancora in crescita) che continuerà a lavorare in piena trasparenza a fianco di altri collaboratori".

Purtroppo anche tale visione era piuttosto ottimistica: il ritardo era ancora maggiore di quanto analizzato. Ciononostante, il ritardo era ormai chiaro a tutti e fonte di non poco imbarazzo da parte di Oracle. Nei vari blog circolò anche la notizia che il team Oracle fosse in grado di generare bug addirittura durante l'esercizio di sostituzione delle informazioni di copyright di Sun Microsystems per sostituirle con i vari logo Oracle.

Tutto ciò, unito alle infinite discussioni soprattutto interne ai progetti Jigsaw e Lambda, spinse a pensare a una soluzione alternativa diventata "famosa" (o meglio "famigerata") con il nome di "piano B" che prevedeva:

- il rilascio di JDK 7 entro Q2 2011 al netto dei progetti Lambda, Jigsaw, e di una parte minore di Coin;
- la posposizione dei progetti più controversi (Lambda, Jigsaw, e parte di Coin) alla versione Java SE 8, prevista per la fine del 2012.

Come poi si è avuto modo di vedere, anche queste scadenze erano alquanto ottimistiche, altro tipico difetto del personale informatico. È notizia di questi giorni (cfr. [Reinhold, 2012]) che il progetto Jigsaw verrà molto probabilmente fatto slittare addirittura alla versione Java SE 9, attualmente programmata per il 2015.

Vantaggi del piano B

L'obiettivo del piano B era di evitare un'attesa eccessiva, prevista fino al 2013, per il rilascio di Java SE 7, che avrebbe significato ben sette anni dopo il rilascio della precedente major release, Java SE 6. La presentazione del piano B, come era lecito attendersi, suscitò ulteriori dibattiti tra i fautori di tale piano e coloro che invece volevano proseguire con il piano originale. In particolare, i fautori del piano B, utilizzavano i seguenti argomenti a favore della nuova strategia (cfr. [Heiss, 2011]):

- avere una nuova versione stabile facilmente gestibile, anche se di contenuti ridotti, era preferibile rispetto ad avere una versione con molte importanti variazioni, ma solo dopo un periodo prolungato;

- release più frequenti fornirebbero un maggiore grado di flessibilità ai clienti permettendo una migliore e più graduale programmazione dell'integrazione delle nuove feature;
- con una versione "intermedia" l'evoluzione del linguaggio sarebbe stata più fluida, grazie a feedback immediati, mirati e più facilmente gestibili; una release "definitiva" dopo tanti anni avrebbe incorporato una moltitudine di funzioni e quindi di potenziali bug;
- le aziende software sembrerebbero non gradire particolarmente approcci del tipo "let's wait and ship everything" ("attendiamo e consegnammo tutto");
- cambiamenti non eccessivamente radicali, oppure anche cambiamenti radicali, ma in numero molto limitato, avrebbero favorito il tasso di adozione;
- il lavoro svolto fino a quel momento, soprattutto nell'area del progetto Coin e l'aggiornamento di NIO 2 già di per se' avrebbe giustificato una nuova release;
- il rilascio anticipato della versione Java SE 7 avrebbe generato una vitale "boccata di ossigeno" agli addetti ai lavori dei progetti Lambda e Jigsaw, allentando la pressione sugli sviluppatori;
- rilasci più frequenti avrebbero permesso di diminuire e gestire meglio i fattori di rischio.

Svantaggi del piano B

A questi vantaggi, i detrattori del piano B, contrapponevano essenzialmente le seguenti motivazioni:

- il grado di adozione della nuova versione poteva non essere così elevato come atteso: dal momento che la versione Java SE 8 avrebbe dovuto seguire a breve termine la Java SE 7, probabilmente molte aziende avrebbero finito per optare all'integrazione diretta di Java SE 8, saltando a piè pari la precedente release;
- Java SE 7 veniva percepita come una versione senza sufficienti feature per essere considerata "major";
- il piano originario A offriva una singola e solida versione invece di consegnare nuove feature a piccoli frammenti, il che doveva semplificare il processo di pianificazione e implementazione dell'adozione da parte delle aziende.

Dall'analisi delle precedenti argomentazioni è possibile evidenziare come i medesimi argomenti venivano utilizzati contemporaneamente sia in quanto punti a favore e che come obiezioni contrarie al piano B! Verosimilmente, Pirandello docet anche nei progetti informatici. Alla fine, come è ben noto, si decise di dar luogo al piano B: ciò nonostante, le date per le due release hanno comunque subito ulteriori importanti dilazioni.

Breve elenco delle nuove feature

Le principali feature introdotte con Java SE 7 sono quelle riportate di seguito.

Progetto Coin ("moneta")

Il progetto Coin prevede una serie di nuove caratteristiche, che riportiamo opportunamente raggruppate per tematica:

- **incremento della consistenza e della chiarezza** ottenuto attraverso l'estensione del costrutto switch per permettere l'utilizzo di stringhe ed il miglioramento dei valori letterali (literal) che prevede l'aggiunta dei valori binari e la possibilità di utilizzare i caratteri sottolineato;
- **semplificazione dell'utilizzo dei generics** basata sulla notazione del diamante che forza il compilatore ad eseguire l'inferenza del tipo e sul miglioramento dei warning relativi all'utilizzo di parametri formali non refiable con la notazione varargs;
- **snellimento della gestione degli errori** resa più concisa grazie alla possibilità di utilizzare catch multipli (ossia di catturare diverse eccezioni con un unico catch), all'introduzione di meccanismi che permetto di aumentare la precisione nel rilancio delle eccezioni, all'implementazione di un nuova versione del **try...catch** che permette di dichiarare le risorse utilizzate all'interno del costrutto (**try-with-resource**).

Da notare che il progetto Coin, almeno secondo i piani iniziali, era molto più ambizioso e includeva altre feature che poi sono state rinviate a **Java SE 8**. Tra queste nuove caratteristiche figurano:

- collezioni di **literals**;
- ripetizione di **annotazioni**, che permette di migliorare la leggibilità del codice sorgente grazie all'applicazione di più istanze di annotazioni dello stesso tipo a un elemento del programma, [Darcy 2011];
- nome dei **parametri** a tempo di esecuzione

Alle nuove funzioni appena citate, Java SE 8 dovrebbe aggiungere tutta una serie di miglioramenti per alcune delle feature di Coin appena introdotte in Java SE 7.

Miglioramento della concorrenza (java.util.concurrent)

Il package **java.util.concurrent** subisce un miglioramento dovuto essenzialmente all'introduzione di quanto riportiamo di seguito:

- framework **Fork/Join** (strumento molto flessibile ed utile per l'implementazione di algoritmi *divide et impera*);

- classe **Phaser** (un'evoluzione del concetto di barriera);
- **ThreadLocalRandom** (generatore concorrente di numeri randomici);
- coda bloccante **LinkedTransferQueue** (interfaccia **TransferQueue**).

API NIO versione 2

In Java SE 7 viene rilasciata la versione 2 dell'API NIO, le cui principali innovazioni sono condensate nei seguenti nuovi package:

- **java.nio.file**, che fornisce una API completa e potente per la gestione dei file/directory, per la gestione dei link simbolici, con inclusa una serie di servizi avanzati quali il **Path Matcher**, il **WatchService**, e così via);
- **java.nio.file.attribute**, che definisce la libreria disegnata per accedere agli attributi dei file e del file system;
- **java.nio.file.spi**, che è il package utilizzato per l'implementazione dei servizi definiti dalla nuova API diversi da quella di default;
- **com.sun.nio.sctp**, che rappresenta il supporto al protocollo Stream Control Transmission Procotol (SCTP).

Aggiornamenti alla API Swing e Java 2D

Per quanto riguarda Swing e Java 2D, ecco le novità:

- **Nimbus**. Look&feel che sostituisce il precedente Metal LoF. Introdotto con Java SE6, con Java SE7 diviene parte di **com.sun.java.swing**.
- **JLayer**. Si tratta di un framework molto potente che permette di variare il look dei componenti e il modo in cui si comportano.
- Standardizzazione **translucent windows**. Si possono creare delle finestre trasparenti.
- **Finestre non rettangolari**. Adesso si possono usare anche finestre di forma diversa dal rettangolo.
- Supporto per delle **font OpenType/CFF**.
- **XRender per Java 2D**. Nuova XRender pipeline, basata su Java 2D, per i nuovi desktop X11.

Internazionalizzazione

Anche il supporto all'internazionalizzazione viene migliorato.

- aggiornamento dello standard **Unicode** alla **versione 6.0**;
- supporto per l'**Extensible Currency Codes** (ISO 4217)
- evoluzione del **Locale** con suddivisione delle responsabilità: formattazione e visualizzazione;

- **NumericShaper**, che è una classe in grado di convertire le cifre da Latin-1 ad altri formati Unicode;
- supporto Unicode 6 nelle **espressioni regolari**.

JDBC 4.1

JDBC 4.1 include sostanzialmente le seguenti caratteristiche:

- possibilità di utilizzare le risorse **Connection**, **ResultSet**, e **Statement** nel costrutto **try-with-resource** introdotto con il progetto Coin;
- l'introduzione dell'interfaccia **RowSetFactory** e della classe **RowSetProvider**, che permettono di creare i vari **RowSet** specifici.

Sicurezza

Per quanto riguarda la sicurezza, assistiamo all'introduzione:

- dell'algoritmo **Elliptic Curve Cryptography** (ECC, Crittografia basata sulle curve ellettiche);
- del supporto per il **Transport Layer Secuirty** (TLS), versioni 1.1 e 1.2.

Aggiornamento dello stack XML

Viene compiuto un adeguato aggiornamento dei componenti dello stack XML alle versioni più recenti e stabili, vale a dire **JAXP 1.4.5**, **JAX-WS 2.2.4**, **JAXB 2.2.3**.

Novità per la JVM

Anche la Java Virtual Machine viene interessata da alcuni cambiamenti:

- estensione della JVM per il supporto dei linguaggi dinamici (nuova istruzione: **invokedynamic**);
- miglioramento del meccanismo di intercettamento e risoluzione di **deadlock**;
- nuova versione del **Concurrent Mark-Sweep Collector** denominato Garbage-first collector.

Gestione JNLP

C'è anche un aggiornamento per la gestione del Java Network Launching Protocol (JNLP).

Java DB Derby

Parallelamente, anche il Java DB Derby subisce degli aggiornamenti.

Conclusioni

In questo capitolo abbiamo presentato una veloce panoramica di Java SE 7.0: un'importante major release per diversi punti di vista, non sempre di carattere tecnico. La versione 7 è importante in primo luogo perché segue il rilascio open source dell'**OpenJDK**. Tale apertura tuttavia ha escluso le Java Class Library. In secondo luogo, dobbiamo guardare con attenzione alla versione 7 perché si è trattato della **prima versione gestita da Oracle** dopo l'acquisizione di Sun Microsystems. Anche se la piattaforma Java è da tempo (febbraio 2002, J2SE 1.4) gestita da un'apposita community abbastanza indipendente, attraverso processi aperti, sebbene molti sanno bene che Sun Micorsystems prima, e Oracle poi hanno sempre esercitato un certo controllo sullo sviluppo di questa tecnologia sempre più strategica per molte aziende. Non a caso nel dicembre 2010 Apache Software Foundation, fucina di idee innovative e framework per la tecnologia Java, decise di dimettersi dal JCP Executive Commette in aperta polemica con l'egemonia esercitata da Sun.

Java SE 7.0 è stata rilasciata **ben dopo 5 anni** dalla precedente versione (tempi biblici per l'informatica) dopo uno sfoltimento significativo dello scope iniziale che comprendeva sia le Lambda expression sia il progetto Jigsaw entrambi inizialmente demandati a Java SE 8. Il progetto Jigsaw di recente è stato addirittura fatto slittare a Java SE 9.

Java SE 7.0 include una serie di cambiamenti che investono diverse aree a partire dallo stesso linguaggio di programmazione (progetto Coin), alle API NIO (rilascio di NIO2), al multi-threading, all'internalizzazione, alla UI, alla sicurezza e così via. Tutti i cambiamenti sono presentati in dettaglio nei capitoli successivi.

Java SE 7 doveva rappresentare una nuova "rivoluzione" del linguaggio Java, ma per via di un acceso dibattito all'interno della comunità si è saggiamente deciso di posticipare alcune feature alla versione 8 (attuazione del famoso piano B) per non ritardare ulteriormente il rilascio della release Java SE 7.

I problemi sorti all'interno dei progetti Lambda e Jigsaw evidenziano quelli che possono essere considerati gli effetti collaterali della gestione del disegno di soluzioni per "consenso allargato" (e non sempre guidato da un'agenda prettamente tecnica). Se da un lato valutare soluzioni tecniche attraverso diversi punti di vista ha la potenzialità di migliorarne il disegno, permettendo di valutare anche aspetti non considerati inizialmente, quando il consenso coinvolge un insieme allargato di tecnici, allora si corre il rischio di dilatare eccessivamente la tempistica e quindi di generare molta frustrazione (ogni singolo punto viene discusso all'infinito) e, in casi estremi, di approdare a soluzioni tecnicamente povere, realizzate al solo scopo di raggiungere il compromesso necessario per uscire da situazioni di ristagno.

Capitolo 2
La "moneta" dei cambiamenti

Introduzione

Questo capitolo è dedicato alla presentazione delle nuove feature sviluppate attraverso il progetto "Coin" ("Moneta") la cui missione ne esprime i driver in maniera eloquente:

"*Project Coin is a suite of language and library changes to make things programmers do everyday easier.*" [Smith, 2011], ("Il progetto Coin include un insieme di cambiamenti del linguaggio e delle API, implementati al fine di semplificare il lavoro che gli sviluppatori svolgono quotidianamente").

Più precisamente, il progetto "Coin" mirava essenzialmente a modificare il linguaggio di programmazione Java. Tuttavia, alcune feature, quali per esempio il **try-with-resource**, come illustrato di seguito, hanno richiesto l'aggiornamento di molteplici classi della Java API (in sostanza tutte le classi che gestiscono una risorsa). Quindi, la modifica della API Java è stata più un effetto che una causa.

La missione del progetto si è tradotta nei seguenti obiettivi:

- rimozione di codice supplementare, extra, al fine di rendere il codice più leggibile;
- modifiche al linguaggio che consentono di rendere il codice più robusto;
- completo allineamento delle modifiche con le versioni passate e le nuove feature.

Altro elemento importante da tener presente è che il progetto "Coin" includeva una serie di importanti vincoli, tra cui i principali sono:

- nessun cambiamento alla JVM, il che significa che le nuove feature non dovevano assolutamente interessare il bytecode;
- solo variazioni minori, quindi nessun nuovo costrutto poteva essere introdotto e nessuno esistente poteva essere rimosso: erano consentite solo alterazioni "minori" alla semantica dei costrutti esistenti;
- nessuna modifica al Java type system (molto brevemente con i termini type system si includono molteplici concetti fondamentali di un linguaggio, quali i tipi base come int, le classi ed interfacce, il tipo null, l'ereditarietà, etc.)

Il progetto Coin inizialmente era più ambizioso ed oltre le nuove feature confluite in Java SE7 ne includeva altre interessanti (come per esempio collezioni di literals, ripetizione di annotazioni, nome dei parametri a tempo di esecuzione). Queste, per questioni di tempo sono state rimosse e, con molta probabilità, confluiranno in Java SE 8.

Incremento della consistenza e della chiarezza

Questo paragrafo presenta due aggiornamenti di sintassi, lo switch con stringhe e l'evoluzione nell'utilizzo dei literal.

Switch con stringhe

Originariamente i costrutti **switch** potevano valutare esclusivamente tipi di dato primitivi quali: **byte**, **short**, **char** e **int**. Con l'introduzione degli **enumeration** (J2SE 5.0, JSR 201, settembre 2004, cfr. [JSR 201, 2004]) il costrutto **switch** è stato esteso per poter valutare anche i **literal** che costituiscono gli **enumeration**, come mostrato nel listato seguente .

```
public class EnumerationSwitchSample {

  // ----------------- CONSTANTS SECTION ----------------
  /** logger */
  private static final Logger LOGGER =
       Logger.getLogger(EnumerationSwitchSample.class);

  // ---------------- ATTRIBUTES SECTION ----------------
  /** day of the week */
  private DayOfTheWeek dayOfWeek = null;

  // ----------------- METHODS SECTION ------------------
  /**
   * Constructor method
   * @param aDay  a day of the week
   */
  public EnumerationSwitchSample(DayOfTheWeek aDay) {
    this.dayOfWeek = aDay;
  }

  /**
```

```
 * @return a comment related to the specific day
 */
public String tellItLikeItIs() {
  String result = null;
  switch (dayOfWeek) {
    case MONDAY:
      result =“Mondays are horrible!”;
      break;

    case FRIDAY:
      result = “Fridays are good.”;
      break;

    case SATURDAY:
    case SUNDAY:
      result = “Weekends are best.”;
      break;

    default:
      result = “Midweek days are so-so.”;
      break;
  }
  return result;
}


// ---------------------- INNER CLASS ------------------
/**
 * This defines an enumeration that represents
 * the days of the week
 */
public static enum DayOfTheWeek {
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY,
```

```
    SUNDAY
  }

  /**
   * Main method used to test the class: bad practice!
   * @param args  list of arguments
   */
  public static void main(String[] args) {

    for (DayOfTheWeek aDayOfTheWeek :
         EnumerationSwitchSample.DayOfTheWeek.values()){

      EnumerationSwitchSample aDay =
        new EnumerationSwitchSample(aDayOfTheWeek);
      LOGGER.info("Day of the week:" +aDayOfTheWeek+"..."+
            aDay.tellItLikeItIs());
    }
  }
```

Listato 1. *Esempio di costrutto switch con tipo enumerato.*

L'esecuzione del listato appena mostrato, al netto delle informazioni del log, genera il seguente output:

```
Day of the week:MONDAY...Mondays are horrible!
Day of the week:TUESDAY...Midweek days are so-so.
Day of the week:WEDNESDAY...Midweek days are so-so.
Day of the week:THURSDAY...Midweek days are so-so.
Day of the week:FRIDAY...Fridays are good.
Day of the week:SATURDAY...Weekends are best.
Day of the week:SUNDAY...Weekends are best.
```

L'ulteriore estensione Java SE 7 include la possibilità di eseguire la **valutazione di stringhe** come mostrato nel listato seguente:

```
public class StringSwitchSample {

  // --------------- CONSTANTS SECTION ---------------
```

```
/** logger */
private static final Logger LOGGER   =
    Logger.getLogger(StringSwitchSample.class);

// --------------- ATTRIBUTES SECTION ---------------

// ----------------- METHODS SECTION -----------------
/**
 * return a comment related to the given day of the week
 * @param dayOfWeek a day of the week
 * @return a description of the specific day
 */
public static String tellItLikeItIs(String dayOfWeek) {
  if (dayOfWeek == null) {
    return null;
  }

  String result = null;
  String day = dayOfWeek.toUpperCase().trim();

  switch (day) {
    case “MONDAY”:
      result =“Mondays are horrible!”;
    break;

    case “TUESDAY”:
    case “WEDNESDAY”:
    case “THURSDAY”:
      result = “Midweek days are so-so.”;
    break;

    case “FRIDAY”:
      result = “Fridays are good.”;
    break;

    case “SATURDAY”:
    case “SUNDAY”:
      result = “Weekends are best.”;
```

```
      break;

      default:
        result = null;
      break;
    }

    return result;
  }


  /**
   * Main method used to test the class: bad practice!
   * @param args  list of arguments
   */
  public static void main(String[] args) {
    String[] daysOfWeek = {
      "MONDAY", "TUESDAY",  "WEDNESDAY", "THURSDAY",
      "FRIDAY", "SATURDAY", "SUNDAY"
    };

    for (String aDayOfTheWeek : daysOfWeek ) {
      LOGGER.info("Day of the week:"+aDayOfTheWeek+
            "..."+StringSwitchSample.
tellItLikeItIs(aDayOfTheWeek));
    }
  }
}
```

Listato 2. *Esempio di costrutto switch con stringhe.*

L'output è ovviamente lo stesso, tuttavia, come si può notare, la nuova versione che sfrutta lo switch con stringhe suscita qualche perplessità. Pertanto, sebbene esistano degli scenari in cui il costrutto Switch con stringhe può fornire dei vantaggi, il suo utilizzo in molti casi dovrebbe essere scoraggiato rispetto alla versione più appropriata basata sul tipo enumerator. Una delle motivazioni alla base che porta a preferire i tipi enumerati è che questi permettono di raggruppare le

varie "stringhe" (literals) che lo compongono, trattarle come un oggetto, utilizzare un insieme di metodi standard, e così via. L'utilizzo di una mera lista di stringhe fa sì che si perdano tutti i vantaggi suddetti e si ritorni ad una gestione non molto tipizzata e poco strutturata. Problemi che hanno portato alla definizione del tipo enumerato.

Miglioramento dei Literal: valori binari

Un'altra novità introdotta con Java SE 7 è la possibilità di specificare per i tipi interi **byte**, **short**, **int** e **long**, valori espressi utilizzando il sistema numerico binario. A tal fine è necessario aggiungere il prefisso **0b** o **0B** al numero binario, come mostrato di seguito:

```
byte aByte   = (byte)0b00100001;
short aShort = (short)0b1010000101000101;

int anInt1 = 0b10100001010001011010000101000101;
int anInt2 = 0b101;
int anInt3 = 0B101;

long aLong =
0b1010000101000101101000010100010110100001010001011010000101000101
1L;
```

Listato 3. *Esempi di utilizzo della rappresentazione binaria.*

Un'importante avvertenza consiste nel ricordare che in Java il tipo **byte** è **signed**. Ciò comporta che il primo bit, quello più significativo (msb: "the most significant bit") è utilizzato per specificare il segno. Quindi bisogna porre attenzione a dichiarazioni del tipo (**byte**)**0b10100001** che generano un valore uguale a "**-95**".

La possibilità di poter specificare numeri in rappresentazione binaria può risultare molto utile qualora sia necessario eseguire degli interfacciamenti con driver/componenti a basso livello, sia quando si vuole gestire dei byte attraverso opportune maschere binarie.

Miglioramento dei Literal: introduzione degli underscore

Altro miglioramento dei literal è la possibilità di includere un numero qualsiasi di **caratteri di sottolineatura** (_) tra le cifre di un valore letterale numerico.

Ciò permette di raggruppare cifre in valori numerici, che possono migliorare la leggibilità del codice. Per esempio, se il codice contiene numeri con molte cifre, è possibile utilizzare un carattere di sottolineatura per **separare le cifre in gruppi di tre**, in modo simile a come si userebbe un punto delle migliaia. Di seguito sono riportati alcuni modi possibili di utilizzare il carattere di sottolineatura in letterali numerici:

```
long creditCardNumber   = 1234_5678_9012_3456L;
long socialSecurityNumber  = 999_99_9999L;
float pi    = 3.14_15F;
long hexBytes     = 0xFF_EC_DE_5E;
long hexWords     = 0xCAFE_BABE;
long maxLong    = 0x7fff_ffff_ffff_ffffL;
byte nybbles    = 0b0010_0101;
long bytes    = 0b11010010_01101001_10010100_10010010;
```

Listato 4. *Esempi di utilizzo del carattere sottolineato nei valori numerici.*

Da notare che è possibile inserire caratteri underscore solo tra le cifre, mentre non è possibile inserirli nelle seguenti parti:

- all'inizio o alla fine di un numero
- accanto a un punto decimale in una costante in virgola mobile
- prima di un suffisso **F** o **L**
- nelle posizioni in cui ci si aspetta una stringa di digits

Qui di seguito sono riportati esempi che generano errori di compilazione:

```
// Errore: underscore vicino al punto decimale
float pi1 = 3_.1415F;
float pi2 = 3._1415F;

// Errore: underscore prima del suffisso
long socialSecurityNumber1 = 999_99_9999_L;

// Errore: _52 è un nome valido di variabile
int x1 = _52;
```

```
// Errore: non si può inserire un underscore alla fine
int x3 = 52_;
// Errore: underscore nel prefisso 0x
int x5 = 0_x52;
// Errore: underscore all'inizio di un numero
int x6 = 0x_52;
// Errore: underscore alla fine di un numero
int x8 = 0x52_;
// Errore: underscore alla fine di un numero
int x11 = 052_;
```

Listato 5. *Esempi di utilizzo scorretto del carattere sottolineato.*

Di seguito è illustrato un esempio ripreso da un’idea di Josh Bloch (capo architetto Java presso Google).

```
public class Agent007 {
  private static int agent =
          ___00_________00________0000000__+
         __0__0_______0__0____________0__+
        __0__0_______0__0___________0__+
       __0__0_______0__0__________0__+
      __0__0_______0__0_________0__+
     ___00_________00________07_;

  public static void main(String[] args) {
    System.out.println(agent);
  }
}
```

Listato 6. *Semplice gioco con i caratteri sottolineato.*

La possibilità di utilizzare il carattere sottolineato si dimostra particolarmente gradita in tutti i programmi in cui è necessario manipolare valori numerici di una certa entità, come per esempio applicazioni finanziarie. In questo

contesto, questa caratteristica si applica sia nella definizione di costanti numeriche sia nella fase di test, dove è richiesto di impostare e verificare molteplici valori numeri.

Semplificazione nell'utilizzo dei generics

Un altro diamante

Fin dalla prima versione Java ha avuto a che fare con i "diamanti", tuttavia in questo contesto il nome si deve alla figura di rombo (chiamata appunto "diamante" in inglese) formata dai parametri di tipo senza parametri (parentesi agolare aperta e chiusa, <>) che serve per specificare il tipo delle classi generic.

L'idea alla base di questa innovazione è relativa al fatto che in molti casi **il compilatore** è in grado di **dedurre** il **corretto tipo** degli **argomenti** da utilizzare per invocare il costruttore di una classe generica senza doverli scrivere esplicitamente nel codice. Tale deduzione può avvenire attraverso l'esame dell'utilizzo dei relativi oggetti. Questa operazione di **deduzione** eseguita dal **compilatore** si chiama inferenza di tipo (**type inference**). Si consideri per esempio, una dichiarazione del tipo:

```
Map<String, List<String>> myMap = new HashMap<String,
List<String>>();
```

In Java SE 7 essa può essere dichiarata semplicemente sostituendo ai tipi parametrizzati il famoso diamante:

```
Map<String, List<String>> myMap = new HashMap<>();
```

Chiaramente bisogna porre attenzione ad includere le parentesi angolari, in quando una dichiarazione del genere **myMap = new HashMap()** avrebbe un significato ben diverso: si finirebbe per dichiarare una collezione come si soleva fare prima dell'avvento dei generics, definita di tipo grezzo (raw type).

Il compilatore, come specificato sopra, è in grado di eseguire l'inferenza di tipo per la creazione di istanze generiche solo nei casi in cui il tipo di parametri del costruttore sia evidente dal contesto. Si consideri per esempio il seguente listato in cui il compilatore non è in grado di eseguire una corretta inferenza di tipo nella terza istruzione:

```
List<String> list = new ArrayList<>();
list.add("A");
```

```
list.addAll(new ArrayList<>());
```

Il metodo **addAll** genera un errore di compilazione giacché il metodo richiede un parametro di tipo **Collection<? extends String>** mentre il compilatore, non avendo riferimenti, non può far altro che creare un **ArrayList<Object>**.

Sebbene l'inferenza di tipo possa essere utilizzata anche in casi diversi rispetto la sola creazione di oggetti, è consigliabile limitarne l'utilizzo a questi casi, onde evitare confusione. In particolare, in Java è possibile dichiarare parametri di tipo nella firma dei metodi e dei costruttori per creare metodi generici e costruttori generici. Si tratta di una strategia simile alla dichiarazione di un tipo generico, con la differenza che la portata del parametro di tipo è limitata al solo metodo/costruttore in cui viene dichiarata.

Si consideri il seguente listato:

```
public class TypeInferenceSample<T> {

  // ----------------- CONSTANTS SECTION -----------------
  /** logger */
  private static final Logger LOGGER   =
              Logger.getLogger(TypeInferenceSample.class);


  // ----------------- ATTRIBUTES SECTION -----------------
  private T attributeOfGenericType = null;

  // --------------- -- METHODS SECTION ------------------

  /**
   * Set the specific object
   * @param attributeOfGenericType
   *                    an instance of the specific type T
   */
  public void set(T attributeOfGenericType) {
    this.attributeOfGenericType = attributeOfGenericType;
  }

  /**
   * Get the instance of the specific object
```

```
    * @return an object of a generic type
    */
   public T get() {
     return attributeOfGenericType;
   }


   /**
    * Inspect and log the two generic types:
    * the class one and the method one
    * @param u  an instance of the method generic type
    */
   public <U> void inspect(U u) {
     LOGGER.info("T: " + attributeOfGenericType.getClass().
getName());
     LOGGER.info("U: " + u.getClass().getName());
   }

   /**
    * A bad practice using main to test.
    * @param args command line parameters
    */
   public static void main(String[] args) {
     TypeInferenceSample<Integer> integerBox = new
TypeInferenceSample<>();
     integerBox.set(new Integer(10));
     integerBox.inspect("Life is beatiful!");
   }
}
```

Listato 7. *Semplice esempio che mostra il concetto dell'inferenza di tipo.*

Come si può notare, si tratta di una classe che incapsula una variabile di tipo generico. Fin qui tutto normale. La peculiarità di questa classe è rappresentata dalla presenza del metodo **inspect** il quale include nella propria firma un ulteriore tipo variabile <**U**>. Pertanto, come mostrato dal corpo del main, è possibile invocare questo metodo specificando una variabile di tipo diverso, che in

questo caso è una stringa. Per la precisione, l'invocazione del metodo non specifica il tipo stringa, il quale viene dedotto dal compilatore grazie al processo di inferenza di tipo. Come è lecito attendersi, l'output prodotto dall'invocazione del metodo main è:

```
T: java.lang.Integer
U: java.lang.String
```

Miglioramento dei warning e degli errori nell'utilizzo di tipi non reifiable con i varargs

La maggior parte dei tipi parametrizzati, come **ArrayList<Number>** e **List<String>, sono definiti non-reifiable**. Cosa significa? Significa che il **compilatore rimuove le informazioni del tipo**, applicando il processo chiamato **erasure** (cancellazione), e quindi queste non sono più disponibili in fase di esecuzione. Questa strategia, ampliamente utilizzata da J2SE 5.0 è stata necessaria per assicurare la compatibilità binaria con le librerie Java e le applicazioni create prima dell'introduzione dei generics. Poiché, in fase di compilazione, il processo di erasure rimuove le informazioni da tipi di parametri, questi tipi non sono **reifiable**.

L'inquinamento dell'heap (**heap pollution**), si verifica quando una variabile di un tipo parametrizzato si riferisce a un oggetto che non è di quel tipo parametrizzato. Questa situazione può verificarsi solo se il programma ha eseguito qualche operazione che darebbe luogo a un **unchecked warning** a **tempo di compilazione**. Questo tipo di warning è generato se, al momento della compilazione o in fase di runtime, la correttezza di un'operazione che coinvolge tipi parametrizzati non può essere verificata. Si consideri il seguente frammento di codice:

```
List l = new ArrayList<Number>();
List<String> ls = l;          // unchecked warning
l.add(0, new Integer(42));    // unchecked warning
String s = ls.get(0);         // ClassCastException!
```

In fase di compilazione, l'**ArrayList<Number>** e la **List<String>** perdono le informazioni di tipo, quindi diventano, rispettivamente, **ArrayList** e **List**. Pertanto quando si cerca di assegnare la lista **l** a **ls**, il compilatore genera un **unchecked warning** giacché non è in grado di verificare, e tanto meno lo sarà la JVM, se la variabile **l** punta o meno a una lista di stringhe. In questo caso, poiché **l** punta a una lista di **Number** si genera l'inquinamento dello heap. Una situazione analoga si verifica con l'istruzione successiva. Nel caso del **get** il compilatore non genera

alcun errore in quanto l'istruzione è legittima anche se poi a runtime si riceve un **cast exception** (un qualsiasi programmatore anche junior è in grado di vederne il problema a prima vista). La domanda che potrebbe sorgere è perché il compilatore non generi un errore in uno scenario di inquinamento del tipo... La risposta ancora una volta è **backwards compatibility**: compatibilità con le versioni precedenti.

Ci sono degli errori e dei warning del compiler che si possono verificare quando si usano parametri formali "non reifiable" con metodi **varargs**. Si consideri la seguente classe:

```
public class NewWarningSample {

  public static <T> void addToList(   List<T> listArg,
                  T... elements) {
    for (T x : elements) {
      listArg.add(x);
    }
  }

  /**
   * The faulty method (class cast exception).
   * @param l array of list of strings
   */
  public static void faultyMethod(List<String>... l) {
    Object[] objectArray = l; // ok
    objectArray[0] = Arrays.asList(new Integer(42));
    String s = l[0].get(0); // ClassCastException
  }

  /**
   * A bad practice using main to test.
   * @param args command line parameters
   */
  public static void main(String[] args) {
    List<String> stringListA = new ArrayList<String>();
    List<String> stringListB = new ArrayList<String>();

    NewWarningSample.addToList(
         stringListA, "Seven", "Eight", "Nine");
```

```
    NewWarningSample.addToList(
         stringListA, “Ten”, “Eleven”, “Twelve”);
    List<List<String>> listOfStringLists =
         new ArrayList<List<String>>();
    NewWarningSample.addToList(
      listOfStringLists, stringListA, stringListB);

    NewWarningSample.faultyMethod(
      Arrays.asList(“Hello!”), Arrays.asList(“World!”));
  }
}
```

Listato 8. *Semplice esempio per mostrare i nuovi warning.*

Dalla versione Java SE 7 il compilatore è in grado di generare il seguente warning nei parametri varargs presenti nella firma dei due metodi:

```
warning: [varargs] Possible heap pollution
   from parameterized vararg type T
```

Quando il compilatore esegue il **parsing** di un **metodo varargs**, traduce il **parametro formale varargs** in un **corrispondente array**. Il problema è che Java **non consente** la **costruzione diretta** di **array** di **tipi parametrizzati** (a tal fine è necessario eseguire un work-around, come esempio dichiarare un array di oggetti e quindi farne il casting al tipo parametrizzato). Nel metodo **NewWarningSample.addToList** il compilatore traduce il parametro formale **varargs T... elements** nel corrispondente array parametrizzato **T** [] **elements**. Tuttavia, a causa del processo di erasure, il compilatore è costretto a convertire il parametro formale **varargs** nell’array di oggetti **Object** [] **elements**. Di conseguenza, si creano le basi per **possibili inquinamenti dell’heap**.

Da notare che, come da consuetudine, i warning possono essere soppressi grazie al ricorso alle corrispondenti annotazioni. Questa nuova tipologia di warning può essere soppressa per mezzo della nuova annotazione: **@SafeVarargs**.

Gestione degli errori concisa

Catch multiplo

Il **try-catch** è uno dei costrutti fondamentali Java per la corretta gestione delle risorse e più in generale delle eccezioni.

La struttura classica, prima della versione Java SE 7, prevede la seguente forma canonica:

```
try {
  // istruzioni che possono generare eccezioni
  }
} catch (ExceptionType1 e1) {
  // gestione eccezione tipo 1
} catch (ExceptionType2 e2) {
  // gestione eccezione tipo 2
} finally {
  // possibilità di eseguire operazioni finali,
  // come la corretta chiusura delle risorse
}
```

Listato 9. *Forma canonica del costrutto try-catch pre Java SE 7.*

In questa forma canonica i tentativi di gestione delle eccezioni, nella maggior parte dei casi, si risolvono nell'eseguirne il log e quindi demandarne la gestione ai livelli superiori. Accade raramente che la parte di codice dove viene generata un'eccezione sia la più adatta a gestirla. Normalmente le informazioni necessarie per prendere le opportune contromisure si trovano a qualche livello superiore (questo argomento è trattato accuratamente nel libro [Vetti Tagliati, 2008], capitolo 5). La logica conseguenza è che, nella maggior parte dei casi, le istruzioni utilizzate per gestire l'eccezione di tipo 1 e di tipo 2 sono assolutamente identiche.

In considerazione di tale circostanza e per evitare il pessimo vizio di alcuni sviluppatore di gestire tutti i casi con un catch (Exception e), Java SE 7 offre una **nuova sintassi** che permette di **raggruppare diverse eccezioni** nella stessa clausola **catch**. La logica dietro questa introduzione è evitare che alcuni sviluppatori, per ridurre la quantità di codice da scrivere, ricorrano alla pericolosa tecnica di eseguire direttamente il **catch di Exception**.

Con Java SE 7 la struttura **try-catch-finally** può essere definita come segue:

```
try {
  // istruzioni che possono generare eccezioni
} catch (ExceptionType1 | ExceptionType2 e) {
```

```
    // gestione eccezioni
  } finally {
    // possibilità di eseguire operazioni finale,
    // come la corretta chiusura deglle risorse
  }
```

Listato 10. *Nuova sintassi per il costrutto try-catch.*

Ciò permette di rendere il codice più sintetico e in generale più elegante. Tuttavia è necessario stare attenti a diverse insidie. Un primo esempio è che spesso è necessario poter gestire diverse eccezioni di cui una è specializzazione dell'altra (per esempio **FileNotFoundException** e **IOException**). Chiaramente non è possibile inserire un'eccezione e la sua specializzazione nella stessa riga. Inoltre bisogna porre ben attenzione prima di includere diversi tipi di eccezione in un medesimo **catch** al fatto che diversi tipi di eccezioni potrebbero richiedere gestioni diverse. Come menzionato prima, la speranza è che un costrutto di questo tipo rimuova la cattiva pratica di evitare lunghe **sequence catch** con un catch unico del tipo **catch (Exception e)** . In altre parole, si modifica lo strumento per evitarne un cattivo uso. Ecco un semplice esempio di struttura catch con multiple exception.

```
public class MultiCatchSample<T> {

  // ----------------- CONSTANTS SECTION ----------------
  /** logger */
  private static final Logger LOGGER =
        Logger.getLogger(MultiCatchSample.class);

  // ----------------- ATTRIBUTES SECTION ----------------

  // ----------------- METHODS SECTION ------------------

  /**
   * This method returns the instance of a class
   * given by its name
   * @param className name of the class to use
   *          to generate new instances
```

```
 * @return newly created instance
 *       null in case of problems
 */
public T getGenericInstance(String className) {
  T newInstance = null;

  try {
    newInstance = (T) Class.forName(className).newInstance();

  } catch (ClassNotFoundException e) {
    LOGGER.warn(e);

  } catch (InstantiationException e) {
    LOGGER.warn(e);

  } catch (IllegalAccessException e) {
    LOGGER.warn(e);

  } catch (ClassCastException e) {
    LOGGER.warn(e);
  }

  return newInstance;
}

/**
 * This method returns the instance of a class
 * given by its name
 * @param className name of the class to use
 *          to generate new instances
 * @return newly created instance
 *       null in case of problems
 */
public T getGenericInstance2(String className) {
  T newInstance = null;
  try {
    newInstance = (T) Class.forName(className).newInstance();
  } catch (ClassNotFoundException |
```

```
        InstantiationException |
        IllegalAccessException |
        ClassCastException e) {
      LOGGER.warn(e);
    }
    return newInstance;
  }

  /**
   * Main method used to test the class: bad practice
   * @param args  list of arguments
   */
  public static void main(String[] args) {

    MultiCatchSample<String>
       aMultiCatchSample =  new MultiCatchSample<>();
    String aString =
       aMultiCatchSample.getGenericInstance(
                    “java.lang.String”);
    LOGGER.info(
      “Result:”+aString+
      ” class:”+aString.getClass().getCanonicalName());

    MultiCatchSample<Object>
       anotherMultiCatchSample = new MultiCatchSample<>();
    Object aObject =
      anotherMultiCatchSample.getGenericInstance2(
              Object.class.getCanonicalName());
    LOGGER.info(“Result:”+aObject+
          ” class:”+aObject.getClass().getName()  );
  }
}
```

Listato 11. *Esempio della nuova sintassi del costrutto try-catch.*

Il listato precedente mostra un esempio di un metodo che genera istanze vuote di classe fornitegli per mezzo del percorso della classe. Questo funziona

correttamente solo nei casi in cui la classe preveda un costruttore vuoto. Da notare che qualora la gestione delle diverse eccezioni sia effettivamente lo stesso, la nuova versione del costrutto **catch** permette effettivamente di scrivere codice più conciso ed elegante.

Aumento di precisione nel rilancio delle eccezioni

Con Java SE 7, **il compilatore** esegue un'**analisi più precisa** delle **eccezioni rilanciate** rispetto a quanto accadeva con le versioni precedenti. Ciò consente di dichiarare gli specifici tipi di eccezione nella clausola **throws** di una dichiarazione di metodo.

Si consideri il frammento di codice riportato di seguito. Come si può notare, vengono definite due nuove eccezioni che entrambe estendono la classe base **Exception**. Ora, il costrutto **try** è in grado di lanciare eccezioni dei due tipi; tuttavia, seguendo **una pratica riprovevole**, il blocco **catch** cerca di intercettare tutte le gestioni di tipo **checked** attraverso il costrutto **catch** (**Exception e**).

All'interno dello stesso blocco c'è il rilancio dell'eccezione intercettata che quindi forza il metodo a dichiarare il **throws Exception**, ulteriore pratica riprovevole.

```
public class MorePreciseReThrowSample {

  static class FirstException extends Exception {
  }

  static class SecondException extends Exception {
  }

  public void  rethrowException(String exceptionName)
       throws Exception {
    try {
      if (exceptionName.equals("First")) {
        throw new FirstException();
      } else {
        throw new SecondException();
      }
    } catch (Exception e) {
      throw e;
    }
```

```
    }
  }
```

Listato 12. *Esempio di gestione e rilancio di eccezioni di tipo Exception.*

Con Java SE 7, è possibile ancora dar luogo alla **pratica riprovevole** del **catch (Exception e)** e rilanciare l'eccezione intercettata **throw e**, ma in questo caso è possibile evitare che la firma del metodo venga inquinata da questa brutta pratica. Infatti, poiché le uniche eccezioni che possono essere generate sono **FirstException** e **SecondException**, è possibile dichiarare nella firma del metodo queste due eccezioni e non quella più generale **Exception**, con un aumento della precisione nel rilancio delle eccezioni, come mostrato nel listato seguente:

```
public void rethrowException(String exceptionName)
     throws FirstException, SecondException {
  try {
    // ...istruzioni che lanciano le eccezioni
  } catch (Exception e) {
    throw e;
  }
}
```

Listato 13. *Esempio di rilancio di eccezioni più precise*

Questa feature non sembrerebbe una assolutamente indispensabile, al contrario sembrerebbe un tentativo di rendere lecita, o perlomeno di neutralizzarne l'espansione nel codice, di una pratica riprovevole, come intercettare e rilanciare eccezioni direttamente per mezzo della super classe Exception.

Try-with-resource

Il nuovo costrutto **try-with-resource**, come suggerisce il nome, è un blocco **try** che dichiara una o più risorse, ossia oggetti che devono essere opportunamente chiuse dopo che il programma ne ha terminato l'utilizzo. L'istruzione **try-with-resource** assicura che ogni risorsa sia chiusa correttamente automaticamente alla fine della dichiarazione. L'intenzione di questa feature è di semplificare la scrittura di **codice più sicuro** e **meno a rischio** dal punto di vista di possibili **memory leak**.

Qualsiasi classe che implementa **java.lang.AutoCloseable**, che comprende tutti gli oggetti che implementano **java.io.Closeable**, può essere utilizzata come risorsa del costrutto try-with-resource.

L'interfaccia **AutoCloseable**, introdotta con la versione Java SE 7, dichiara il solo metodo: **void close() throws Exception**. Si tratta ovviamente del metodo che viene invocato automaticamente sugli oggetti dichiarati nel costrutto **try-with-resources**. Come lecito attendersi, si occupa di chiudere correttamente l'oggetto liberando ogni risorsa gestita. Questo metodo può lanciare un'eccezione qualora non riesca a chiudere la risorsa. Sebbene la firma del metodo includa un'eccezione di tipo **Exception** (a questo livello di generalizzazione non si poteva fare di meglio) gli specifici oggetti scatenano eccezioni più specifiche.

Tutte le classi che gestiscono risorse implementano questa interfaccia. Alcuni esempi, giusto per menzionarne i più importanti, sono:

BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter, ByteArrayInputStream, ByteArrayOutputStream, CharArrayReader, CharArrayWriter, CipherInputStream, CipherOutputStream, DatagramChannel, DatagramSocket, DataInputStream, DataOutputStream, DeflaterInputStream, DeflaterOutputStream, DigestInputStream, DigestOutputStream, FileCacheImageInputStream, FileCacheImageOutputStream, FileChannel, FileInputStream, FileLock, FileOutputStream, FileReader, FileSystem, FileWriter, FilterInputStream, FilterOutputStream, FilterReader, FilterWriter, Formatter, ForwardingJavaFileManager, GZIPInputStream, GZIPOutputStream, InputStream, InputStream, InputStream, InputStreamReader, JarFile, JarInputStream, JarOutputStream, LogStream, MemoryCacheImageInputStream, MemoryCacheImageOutputStream, MulticastSocket, ObjectInputStream, ObjectOutputStream, OutputStream, OutputStream, Pipe.SinkChannel, Pipe.SourceChannel, PipedInputStream, PipedOutputStream, PipedReader, PipedWriter, ProgressMonitorInputStream, PushbackInputStream, PushbackReader, RandomAccessFile, Reader, RMIConnectionImpl, RMIConnectionImpl_Stub, RMIConnector, RMIIIOPServerImpl, RMIJRMPServerImpl, RMIServerImpl, Scanner, SelectableChannel, Selector, SequenceInputStream, ServerSocket, ServerSocketChannel, Socket, SocketChannel, SSLServerSocket, SSLSocket, StringBufferInputStream, StringReader, StringWriter, URLClassLoader, Writer, XMLDecoder, XMLEncoder, ZipFile, ZipInputStream, ZipOutputStream.

Si consideri **java.io.BufferedReader**, una delle classi presenti fin dalle prime versioni di Java (JDK 1.1), che implementa le interfacce **Closeable** e

AutoCloseable (per la precisione, implementa anche **Readable** ma questa non è interessante per il contesto corrente). Come tale può essere usata nel nuovo costrutto **try** nel seguente modo:

```
try (BufferedReader br = new BufferedReader(new
FileReader(filePath))) {
```

La precedente istruzione garantisce che l'oggetto **BufferedReader** sia chiuso (ne verrà invocato il metodo **close**) indipendentemente dal fatto che il costrutto **try** sia eseguito normalmente o terminato bruscamente a seguito di un errore. In sostanza, questo statement equivale ad aggiungere la coppia di istruzioni: **if (br != null), br.close()** nella parte del costrutto **finally**. Sebbene questo costrutto dovrebbe sembrare abbastanza chiaro, ci sono i seguenti aspetti da tenere presente:

1. tipicamente sia i metodi all'interno del **try** sia l'invocazione del **close** (**try-with-resource**) possono scatenare delle eccezioni: nel caso in cui ciò avvenga, allora la prima eccezione generata (quella all'interno del try) viene lasciata sopravvivere mentre quella dell'invocazione close viene soppressa; da notare che ciò è possibile grazie ad un altro aggiornamento effettuato con Java SE 7 che consente di aggiungere e reperire le eccezioni soppresse grazie ai metodi **Throwable.addSuppressed** e **Throwable.getSuppressed**.
2. il costrutto **try-with-resource** può ospitare diverse risorse nella clausola **try**. In questo caso, l'invocazione dei metodi di chiusura segue un ordine inverso alla chiamata seguendo una logica a scatole cinesi. La comunicazione delle eccezioni segue la regola di cui al punto 1.

Il seguente listato riporta un esempio di **try-with-resource**.

```
public class TryWithCatchSample {

  // ----------------- CONSTANTS SECTION -----------------
  /** logger */
  private static final Logger LOGGER =
        Logger.getLogger(TryWithCatchSample.class);

  // ----------------- ATTRIBUTES SECTION -----------------

  // ------------------ METHODS  SECTION ------------------
```

```
/**
 * Copy the file content into a string
 * @param filePath   file path to copy
 * @return the string with the file content.
 *      null in case o problems
 * @throws IOException in case a problem occurs
 *            dealing with the given file
 */
public static String getFileContent(String filePath)
        throws IOException {

  if ( (filePath == null) || (filePath.isEmpty()) ) {
    return null;
  }

  StringBuilder strBuilder = new StringBuilder();
  BufferedReader br2 = null;

  try (BufferedReader br =
     new BufferedReader(new FileReader(filePath)); ) {

    br2 = br;

    String nextLine = null;
    while ( (nextLine = br.readLine()) != null) {
      strBuilder.append(nextLine);
    }
  }   // finally no longer needed!

  // -- the following part is used only to demonstrate
  // -- that the underlying BufferedReader is really closed
  try {
    br2.ready();
  } catch (IOException ioe) {
    LOGGER.info(ioe.getMessage());
  }

  return strBuilder.toString();
```

```
  }

  /**
   * Main method used to test the class: bad practice
   * @param args  list of arguments
   */
  public static void main(String[] args) {

    try {
      String content =
        getFileContent(
          “C:\\prj\\java_se_7\\src\\resource\\log4J.properties”);
      LOGGER.info(“content:”+content);

    } catch (IOException ioe) {
      LOGGER.warn(ioe);
    }
  }
}
```

Listato 14. *Esempio della nuova sintassi try-with-resource.*

Il listato precedente non fa altro che copiare in una stringa, che successivamente viene mostrata nel log, il contenuto del file. Da notare che la variabile **BufferedReader br2** è stata introdotta solo al fine di mostrare che dopo il ciclo **try-with-resource** l’oggetto **BufferedReader** viene effettivamente chiuso. Infatti, l’output mostra sia il contenuto del file (**log4J.properties**), sia il messaggio: “**-Stream closed**”.

Ecco un try-with-resource multiplo:

```
/**
 * Copy the entries present in the given zip file
 * into the given output file
 * @param zipFileName   zip file to read
 * @param outputFileName  output file where to write the entries
 * @throws IOException  a serious problem is occurred
 */
```

```
public static void   writeToFileZipFileContents(
                String zipFileName,
                String outputFileName)
        throws IOException {

  if ( (zipFileName == null) || (zipFileName.isEmpty()) ) {
    return;
  }

  if ( (outputFileName == null) || (outputFileName.isEmpty()) ) {
    return;
  }

  Charset charset = StandardCharsets.US_ASCII;
  Path outputFilePath = Paths.get(outputFileName);

  // Open zip file and create output file with
  // try-with-resources statement

  try ( ZipFile zf = new ZipFile(zipFileName);
      BufferedWriter writer =
       Files.newBufferedWriter(outputFilePath, charset)) {

    String newLine = System.getProperty("line.separator");

    // Enumerate each entry
    for (Enumeration<? extends ZipEntry> entries = zf.entries();
       entries.hasMoreElements();) {

      // Get the entry name and write it to the output file
      String zipEntryName =
         entries.nextElement().getName() + newLine;
      writer.write(zipEntryName, 0, zipEntryName.length());
    }
  }
}
```

Listato 15. *Esempio di costrutto try-with-resource con due risorse.*

Conclusione

In questo capitolo abbiamo analizzato le nuove feature introdotte attraverso il progetto Coin ("Moneta") al linguaggio Java e, in ricaduta, alla corrispondente API. Coin, inoltre, includeva una serie di importanti vincoli, tra cui i principali sono: nessun cambiamento alla JVM, solo variazioni minori e nessuna variazione al Java type system.

Anche il progetto Coin era più ambizioso ed oltre le nuove feature confluite in Java SE7 ne includeva altre interessanti che molto probabilmente confluiranno in Java SE 8.

Le nuove feature sono organizzate nei seguenti gruppi:

- Incremento della consistenza e della chiarezza
- Switch con stringhe;
- Introduzione della codifica binaria;
- Possibilità di utilizzo degli underscore nei valori numerici.
- Semplificazione nell'utilizzo degli generics
- Notazione del diamante
- Miglioramento dei warning e degli errori nell'utilizzo di tipi non reifiable con i varargs
- Concisa gestione degli errori
- Catch multiplo
- Aumento di precisione nel rilancio delle eccezioni
- Try-with-resource

Molte di queste feature permettono effettivamente di rendere il codice più conciso, chiaro e robusto. Tuttavia qualche perplessità resta sull'aumento di precisione nel rilancio delle eccezioni e sul catch multiplo. Entrambe queste feature sembrano essere state introdotte per evitare, o perlomeno arginare, una pessima pratica spesso utilizzata dagli sviluppatori: eseguire il catch e/o il throw direttamente di Exception. Si è variato lo strumento per evitarne un pessimo utilizzo.

Capitolo 3
Fork et impera

Introduzione

Architetture multi-core/multi-processore hanno ormai invaso il mercato di largo consumo. Relegate inizialmente al mondo dei server, dominano oramai il mercato dei desktop e laptop, e più recentemente anche quello degli smartphone e dei tablet. Diviene quindi sempre più pressante disporre di software in grado di sfruttare al massimo le potenzialità di queste architetture.

Nel mondo Java ciò si traduce nell'implementare software che la VM, in collaborazione con il sistema operativo, può facilmente distribuire sui vari core/processori disponibili. Vediamo come il framework fork/join, introdotto con Java SE7, vada in questa direzione, permettendo di implementare in maniera veloce ed elegante algoritmi "divide and conquer".

Per dettagli circa la politica multi-threading in Java consultare l'appendice A "Breve storia del multi-threading in Java".

In maniera assolutamente consistente con il disegno del package della concorrenza, una delle feature più apprezzate introdotte con J2SE 5.0, Doug Lea ha sia condotto i lavori (JSR 166) sia scritto le principali classi introdotte con Java SE7.0.

Moore vs. Amdahl

Gordon Moore, uno dei fondatori di Intel, nel 1975 ([Moore, 1975]) giunse alla formulazione della versione definitiva di una legge empirica (divenuta poi famosa con il nome di "Prima Legge di Moore") basata su una serie di osservazioni effettuate presso i laboratori di ricerca e sviluppo (R&D) della Fairchild di cui ne era il direttore: *"Le prestazioni dei processori e il numero dei transistor presenti nei relativi circuiti integrati raddoppiano ogni 24 mesi"* (le prime due versioni di questa legge prevedevano un fattore temporale di 12 e poi di 18 mesi). Questa legge si è dimostrata veritiera per decenni. Per molti si è trattato di una vera e propria auto-profezia: molte aziende cominciarono a credere in questa legge e quindi a utilizzarla come base per le proprie strategie; le aziende manifatturiere hanno prodotto microprocessori sempre più veloci e sempre più complessi.

Nell'ultimo decennio, tuttavia, sembrerebbe che si sia giunto ad un punto di rottura. Viste le oggettive difficoltà di miniaturizzazione dei MOSFET (metal-oxide-semiconductor field-effect transistor, ovvero transistor a effetto di campo basati su semiconduttori ossido di metallo) gli addetti ai lavori hanno cominciato a dubitare che l'evoluzione esponenziale della microelettronica sia ancora sostenibile. Le aziende manifatturiere di circuiti integrati incontrano sempre maggiori difficoltà nella progettazione di nuovi microprocessori per problemi di carattere: fisico, tecnologico, economici e, non ultimi di affidabilità dei dispositivi. Tanto che ormai gli addetti ai lavori sono portati a credere che la tendenza di aumento della complessità dei circuiti molto probabilmente continuerà, mentre la tendenza di incremento della velocità verosimilmente subirà un drastico rallentamento. Le aziende produttrici di processori stanno rispondendo a questo scenario investendo sempre più sull'aumentando del numero di processori presenti nei chip piuttosto che impiegare risorse per aumentare significativamente le performance dei singoli processori. Questa tendenza è seguita dalle aziende hardware e software: dal momento che l'aumento delle prestazioni dei singoli server è limitato e molto oneroso molte aziende cominciano ad orientarsi sulle farm di server su cui far girare software in grado di parallelizzare e distribuire il carico di lavoro. Pertanto si assiste a una **rivalutazione** delle **direttive** dell'**architetto software Gene Amdahl** e della sua legge (cfr. [Amdahl, 1967]) utilizzata per trovare il massimo miglioramento possibile in un algoritmo quando solo una sua parte può essere velocizzata per mezzo di un'attenta parallelizzazione. Questa legge è dunque spesso utilizzata nel calcolo parallelo per prevedere l'aumento di velocità massima teorica in presenza di più processori, il cui limite è dato dal tempo richiesto per l'esecuzione del frammento sequenziale del programma.

Gli studi di Amdahl furono poi ripresi ed estesi da Gustafson [Gustafson, 1988]. Tra i diversi contributi apportati, un punto fondamentale consiste nella revisione di una tesi di Amdahl secondo la quale un determinato problema rimane tale e non varia in funzione delle caratteristiche dei sistemi di esecuzione. Le osservazioni di Gustafson vanno invece nella direzione opposta: gli addetti ai lavori tendono ad utilizzare **approcci diversi**, "**a scalare il problema**" per cercare di ottenere il **massimo dalla potenza di calcolo disponibile**. Questo punto è cruciale in quanto consente di superare diversi limiti al parallelismo stabiliti da Amdahl.

L'obiettivo di questo paragrafo non è tanto di confrontare l'analisi empirica di Moore con gli studi di Amdahl/Gustafson, ma quello di evidenziare che gioco forza il parallelismo è verosimilmente il trend futuro almeno per la prossima decade. In tale direzione vanno alcune analisi eseguite presso la Berkeley University

[Asanovic et al, 2006] che affermano che presto si arriverà a sistemi con oltre 200 core: lunga vita a Amdahl.

Le nuove classi della concorrenza

Tra le nuove classi introdotte con Java SE 7 figura **java.util.concurrent.ThreadLocalRandom** la cui funzione è quella di generare numeri (pseudo)randomici a partire dal seme (**seed**) inizializzato internamente all'atto della costruzione dell'oggetto. Da notare che sebbene **ThreadLocalRandom** disponga di un metodo per l'impostazione esplicita del seme (**setSeed**), la relativa invocazione scatena un'eccezione **UnsupportedOperationException** (il metodo è stato disegnato solo per l'inizializzazione interna gestita per altro dalla classe **Random**).

Da un punto di vista funzionale **ThreadLocalRandom è assolutamente equivalente alla classe java.util.Radom** (il che è abbastanza normale considerato che **ThreadLocalRandom** estende **Radom**).

Per capire la ragione della nuova classe è sufficiente leggere la documentazione JavaDoc della classe **Random**: **le istanze della classe Random sono threadsafe**. Pertanto l'utilizzo della stessa istanza genera accessi contesi e quindi un degrado delle performance. **ThreadLocalRandom** è stata implementata per creare un generatore di numero randomici confinato al corrente thread in esecuzione. Ciò, per esempio, è particolarmente utile quando una serie di task hanno bisogno di utilizzare dei numeri randomici concorrentemente, si pensi per esempio a tipiche simulazioni eseguite per mezzo del modello Monte Carlo che richiede la generazione di decine o centinaia di migliaia di sequenze randomiche. In tale contesto, la nuova classe **permette di eliminare il collo di bottiglia dovuto alla contesa e quindi di migliorare le performance**. L'utilizzo di questa classe prevede il seguente codice:

```
ThreadLocalRandom.current().nextInt()
```

Da notare che è una volta invocato il metodo statico **current**() è poi possibile richiedere un numero randomico in funzione al tipo richiesto attraverso gli specifici metodi: **nextInt**, **nextLong** e **nextDouble**. Questi metodi presentano due versioni (overloading): la prima che prevede come unico parametro il limite superiore e quindi i numeri sono generati nell'intervallo zero incluso, numero specificato escluso; e la seconda che invece richiede di specificare l'intervallo. In questo caso i numeri sono generati dal limite inferiore incluso a quello superiore escluso. Ecco di seguito un semplice esempio di utilizzo di alcuni metodi della class **ThreadLocalRandom**:

```
public class ThreadRandomSample {

  // --------------- CONSTANTS SECTION ---------------
  /** logger */
  private static final Logger LOGGER   =
       Logger.getLogger(ThreadRandomSample.class);

  // --------------- ATTRIBUTES SECTION --------------

  // ---------------- METHODS SECTION ----------------

  public static void main(String[] args) {

    LOGGER.info(
       ThreadLocalRandom.current().nextInt(100));
    LOGGER.info(
       ThreadLocalRandom.current().nextInt(20,40));

    LOGGER.info(
       ThreadLocalRandom.current().nextDouble(100.00));
    LOGGER.info(
       ThreadLocalRandom.current().nextDouble(20.00, 40.00));

    LOGGER.info(
       ThreadLocalRandom.current().nextLong(100));
    LOGGER.info(
       ThreadLocalRandom.current().nextLong(20,40));

    LOGGER.info(
       ThreadLocalRandom.current().nextGaussian());
  }
}
```

Listato 1. *Semplice esempio di utilizzo della classe ThreadLocalRandom.*

Dall'analisi del codice sorgente della classe ThreadLocalRandom (cfr. figura 1) si nota che questa estende la classe java.util.Random (la quale fa uso di servizi

forniti dalla piattaforma di esecuzione per mezzo di sun.misc.unsafe) ed esegue la composizione con ThreadLocal (inizializzato staticamente) per mantenere istanze di sé stessa, ognuna associata esclusivamente ad un Thread che ne fa uso. Se Java avesse incluso l'ereditarietà multipla, questa classe, molto probabilmente, sarebbe stata implementata attraverso tale tipo di ereditarietà.

Nuova coda bloccante

Java SE 7 introduce una nuova coda bloccante denominata **LinkedTransferQueue<E>** che estende la classe astratta AbstractQueue ed implementa la nuova interfaccia **TransferQueue<E>** che, a sua volta, estende l'interfaccia **BlockingQueue<E>** (cfr. figura 2). La nuova interfaccia TransferQueue definisce il comportamento peculiare della nuova coda ossia include la dichiarazione di metodi come: tryTransfer, transfer, hasWaitingConsumer, e così via.

LinkedTransferQueue è stata disegnata esplicitamente per fornire un valido supporto a problemi del tipo produttore/consumatore dove il produttore resta bloccato nell'operazione di inserimento di un nuovo elemento nella coda ("put") fino quando il consumatore effettivamente consuma l'elemento. Si tratta quindi di un ulteriore passo in avanti giacché le implementazioni precedenti si limitavano a bloccare il produttore fino all'effettivo inserimento dell'elemento nella coda. Il nuovo comportamento è evidenziato dal nome dei metodi della nuova interfaccia chiamati appunto "trasferimento": **transfer(E e)** e **tryTransfer(E e)**, proprio per enfatizzare il concetto che il passaggio è completato solo quando

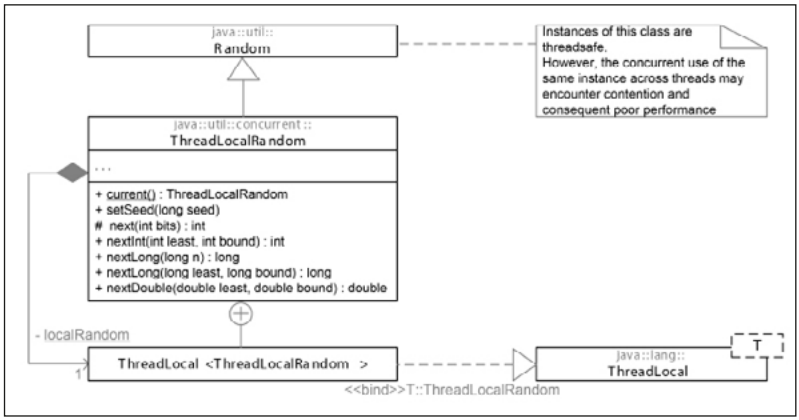


Figura 1. *Class diagram della classe ThreadLocalRandom.*

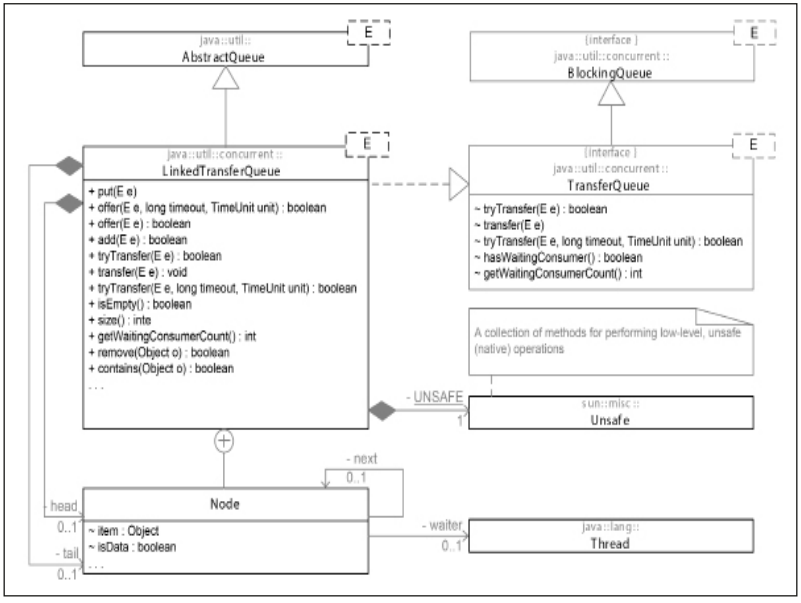


Figura 2. *Class diagram della classe LinkedTransferQueue, la nuova coda bloccante.*

un Thread produttore passa effettivamente l'oggetto a un thread consumatore e quindi avviene il trasferimento. Da notare che **tryTransfer** è la versione non bloccante del metodo **transfer**, dove il metodo è eseguito solo se il passaggio può essere eseguito immediatamente (ciò significa che esiste un thread consumatore pronto) o entro il timeout specificato.

Questa coda, per la precisione, non è totalmente differente da altre, e in particolare dalla **SynchronousQueue<E>** (Java 1.5, la cui nuova implementazione utilizza l'API **transfer**), che è una coda di dimensione **0**. Interrogato circa le motivazioni che hanno spinto a realizzare questa nuova feature, dal momento che comunque già esistono diverse implementazioni di code, Doug Lea ha risposto (cfr. [Lea, 2009]) che esiste una risposta molto semplice. In realtà **avrebbero dovuto dar luogo a questo disegno direttamente nella versione 1.5** piuttosto che implementare le altre code che ora generano una certa confusione. Tuttavia a quei tempi non avevano ancora maturato la visione attuale che si è formata con il senno di poi. Comunque è il classico caso del "meglio tardi che mai".

Le caratteristiche di questa nuova coda sono un superset di quelle fornite da **ConcurrentLinkedQueue**, **SynchronousQueue** (modalità "fair"), e **LinkedBlockingQueues**. Si tratta di una migliore implementazione non soltanto perché è possibile utilizzare le diverse feature contemporaneamente (**put**, **poll**, **transfer**, etc.), ma anche perché lo stesso disegno è stato migliorato e reso decisamente più efficiente (soprattutto grazie all'utilizzo massiccio di servizi nativi, integrati per mezzo della classe: sun.misc.unsafe). Un articolo pubblicato da William Scherer, Doug Lea e Michael Scott (cfr. [Scherer, 2009]) mostra chiaramente che **le performance offerte dalla nuova classe LinkedTransferQueue sono significatamente migliori di quelle offerte dalle precedenti implementazioni**.

Vediamo di seguito un esempio di LinkedTransferQueue:.

```
public class LinkedTransferQueueSample {

  // ------------------- CONSTANTS SECTION ------------------
  /** logger */
  private static final Logger LOGGER =
     Logger.getLogger(LinkedTransferQueueSample.class);

  // ------------------- ATTRIBUTES SECTION -------------------

  // ------------------- METHODS SECTION ------------------

  // ------------------------ INNER CLASSES -------------------

  /**
   * Simple producer
   */
  public static class SimpleProducer implements Runnable {

    TransferQueue<String> transfer = null;

    public SimpleProducer(TransferQueue<String> transfer) {
      this.transfer = transfer;
    }

    @Override
    public void run() {
```

```
    for (int ind = 0; ind < 10; ind++) {
      LOGGER.info(
        "Thread" + Thread.currentThread().getId()+
        " Produced: Element_" + ind);
      try {
        // Wait for a consumer
        transfer.transfer("Element_" + ind);
      } catch (InterruptedException e) {
        LOGGER.info("Ignore exception:" +
               e.getMessage());
      }
    }

    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      LOGGER.info("Ignore exception:" +
             e.getMessage());
    }

    LOGGER.info(
      "Thread" + Thread.currentThread().getId()+
      "Try to transfer 'Extra element'");
    transfer.tryTransfer("Extra element");
    LOGGER.info(
      "Thread" + Thread.currentThread().getId()+
      "Transferered 'Extra element'");
  }

}


/**
 * Simple consumer.
 */
public static class SimpleConsumer implements Runnable {

  TransferQueue<String> transfer = null;
```

```
  public SimpleConsumer(TransferQueue<String> transfer) {
    this.transfer = transfer;
  }


  @Override
  public void run() {
    for (int ind = 0; ind < 11; ind++) {

      try {
        String element = transfer.take();
        LOGGER.info(
          "Thread"+ Thread.currentThread().getId()+
          " Consumed: "+ element);
      } catch (InterruptedException e) {
        LOGGER.info("Ignore exception:" +
              e.getMessage());
      }
    }
  }

}

// Main method
public static void    main(String[] args)
       throws InterruptedException {

  TransferQueue<String> transfer =
          new LinkedTransferQueue<String>();
  SimpleProducer aProducer =
          new SimpleProducer(transfer);
  Thread treadProducer =
          new Thread(aProducer);
  treadProducer.start();

  try {
    Thread.sleep(1_000); // wait a second please
  } catch (InterruptedException e) {
```

```
      LOGGER.info("Ignore exception:" + e.getMessage());
    }

    SimpleConsumer aConsumer = new SimpleConsumer(transfer);
    Thread treadConsumer = new Thread(aConsumer);
    treadConsumer.start();
  }
} // --- end of class ---
```

Listato 2. *Semplice implementazione del pattern producer/consumer basato su LinkedTransferQueue*

Il listato sopra mostra una semplice implementazione del pattern **producer/consumer** con un solo thread che genera i dati da consumare e un solo thread che si occupa di acquisire tali dati. Dall'esecuzione del metodo **main** si osserva che il **producer** tenta di iniziare a produrre le informazioni, più precisamente di trasferirle (**transfer.transfer("Element_"+ind);**) ma viene immediatamente bloccato in quanto non ci sono ancora consumer. Il codice del **main** in effetti introduce un ritardo artificioso di **1 secondo** prima di avviare i consumer, proprio per evidenziare la natura bloccante del metodo. Per finire, il producer, alla fine del suo ciclo di produzione dati, ne produce uno finale (**"Extra element"**) il cui trasferimento avviene per mezzo dell'istruzione non bloccante **tryTransfer.**

Deque e work stealing: una nuova strategia di collaborazione

La più importante delle nuove feature introdotte con Java SE 7 nel package della concorrenza è probabilmente il **framework fork/join**. Questo introduce **una nuova strategia di collaborazione tra thread presenti nel medesimo pool**, nota con il nome di **work stealing** (sottrazione di lavoro) la cui implementazione è basta su una struttura dati denominata **deque**. Per la precisione l'interfaccia **Deque<E>** è stata introdotta con Java SE 6 mentre ciò che è stato aggiunto in Java 7 è l'implementazione **ConcurrentLinkedDeque**. Come verrà descritto nei paragrafi successivi, la peculiarità di questa struttura dati è di far convivere un funzionamento **FIFO** (First In, First Out) tipico delle code, evidenziato dai metodi **addLast(e)**, **offerLast(e)**, **removeFirst()**, **pollFirst()**, **getFirst()**, **peekFirst()** con una strategia **LIFO** (Last In First Out) tipico delle pile, evidenziato dai metodi **addFirst(e)**, **removeFirst()**, **peekFirst()**. Questo doppio funzionamento è utilizzato per far in modo che un Thread possa eseguire il push/pop dei propri

task da eseguire mentre gli altri Thread possano "rubare" i lavori che da più tempo risiedono nella struttura dati. Maggiori informazioni sono riportate di seguito nel contesto del framework fork/join.

L'evoluzione delle barriere: la classe Phaser

Barriere

La versione originale del package della concorrenza include due importanti implementazioni del concetto di **barriera**: **CyclicBarrier** e **CountDownLatch**. Sebbene si tratti di due implementazioni molto interessanti, il loro utilizzo non sempre fornisce un elevato livello di flessibilità e una API altrettanto ricca e potente. Per risolvere queste limitazioni, **Java SE 7** fornisce una nuova classe **java.util.concurrent.Phaser** ("fasatore", in italiano). Si tratta di un valido strumento per implementare complessi processi paralleli che richiedono ai vari Thread coinvolti di sincronizzarsi in diversi punti. Le parti fondamentali di questa classe/framework sono riportate di seguito.

Registrazione

La registrazione rappresenta un primo importante livello di flessibilità. Infatti, a differenza delle altre implementazioni del concetto di barriera, **le istanze Phaser rendono possibile variare nel tempo (dinamicamente) i soggetti interessati alla sincronizzazione**. I partecipanti (questi sono tipicamente, ma non necessariamente, **Thread**) possono essere registrati in qualsiasi momento attraverso i metodi **register()**, **bulkRegister(int)**, o attraverso le apposite varianti del metodo costruttore. In maniera simmetrica, i vari soggetti possono rimuovere la propria registrazione in qualsiasi momento dell'arrivo utilizzando il metodo **arriveAndDeregister()**. Da notare che oggetti **Phaser** gestiscono tre contatori: il numero di soggetti registrati, il numero di soggetti arrivati e quello dei soggetti attesi.

Sincronizzazione

Come anche nel caso della classe **CyclicBarrier**, anche **un'istanza Phaser può essere utilizzata per gestire diverse sincronizzazioni**. Il metodo **arriveAndAwaitAdvance()**, segnala l'arrivo al **Phaser** da parte di un soggetto e attende l'arrivo degli altri, ha lo stesso effetto del metodo **CyclicBarrier.await**. Ogni istanza **Phaser** (e ogni rigenerazione di uno stesso **Phaser**) ha associato un numero di fase che inizia da zero e viene incrementato ogni volta che i vari soggetti arrivano al **Phaser** (ricomincia poi da zero quando raggiuge **Integer.MAX_VALUE**). La sincronizzazione prevede due parti fondamentali: arrivo e attesa.

Arrivo

I metodi **arrive**()e **arriveAndDeregister**() permettono ai soggetti di notificare il proprio arrivo. Si tratta di metodi non bloccanti che restituiscono il numero di arrivo della fase: il numero della fase del **Phaser** per cui l'arrivo è applicato. Quando arriva l'ultimo soggetto atteso da una determinata fase, viene eseguita un'azione opzionale di pre-avanzamento fase. Tale azione è eseguita per mezzo dell'invocazione del metodo **onAdvance(int, int)**. L'override di questo metodo è simile alla fornitura di un'azione-barriera ad un'istanza **CyclicBarrier**, con il vantaggio di essere molto più flessibile.

Attesa

Il metodo **awaitAdvance(int)** richiede un argomento che indica il numero arrivo fase (questo numero è tipicamente quello restituito dai metodi **arrive** o **arriveAndDeregister**), e termina quando il **Phaser** avanza o è già in una fase diversa. A differenza di soluzioni basate sulla classe **CyclicBarrier**, il metodo **awaitAdvance** continua ad attendere, anche qualora il Thread in attesa venga interrotto. Questo metodo prevede anche le versioni interrompibili (**awaitAdvanceInterruptibly(int phase)**) e dotate di timeout (**awaitAdvanceInterruptibly(int phase, long timeout, TimeUnit unit)**), tuttavia eccezioni generate mentre i task si trovano in attesa non cambiano lo stato del **Phaser**. Se necessario, è possibile eseguire operazioni di recovery del blocco di codice di gestione dell'eccezione, spesso dopo aver richiesto il **forceTermination**. I **Phaser** possono essere utilizzati anche da task in esecuzione in un **ForkJoinPool**, allo scopo di assciurare un livello sufficiente di parallelismo per eseguire alcuni task mentre gli altri sono bloccati in attesa dell'avanzamento di una fase.

Termination

Un **Phaser** può entrare in uno stato di terminazione: in tal caso il metodo **isTerminated**() restituisce un valore **true**. Quando ciò avviene tutti metodi di sincronizzazione terminano immediatamente, sbloccando eventuali Thread in attesa, segnalando questa situazione attraverso la restituzione di un valore negativo. Ulteriori tentativi di registrazione, come è lecito attendersi, non generano alcun effetto. Il processo di terminazione è attivato quando una chiamata al metodo **protetto onAdvance**() restituisce il valore **true**. Questo metodo è invocato automaticamente quando il **Phaser** si accinge a lasciare la fase corrente. L'implementazione predefinita restituisce **true** se il numero dei soggetti registrati diviene zero:

```
protected boolean onAdvance(int phase, int registeredParties) {
  return registeredParties == 0;
}
```

Questo metodo dovrebbe essere sovrascritto (override), nelle situazioni in cui il **Phaser** debba eseguire un numero fisso di iterazioni. Per finire, è sempre possibile invocare il metodo **forceTermination**() al fine di provocare una brusca terminazione del **Phaser** con immediato rilascio dei Thread in attesa.

Tiering

Al fine di migliorare il throughtput grazie alla riduzione delle contese, i **Phaser** possono essere **organizzati in strutture ad albero** (in questo caso si dice che sono "tiered", stratificati). In particolar, ogni oggetto Phaser (cfr. Figura 3) possiede un riferimento al Phaser root (**private final Phaser root**) e a quello genitore (**private final Phaser parent**). Questa organizzazione presenta importanti vantaggi nelle situazioni di **Phaser** con un gran numero di soggetti registrati che, inevitabilmente, finirebbero per risentire di notevoli riduzioni delle performance per via delle contese di sincronizzazione. In tal caso è opportuno organizzarne la struttura in gruppi di **sub-Phaser** che condividano un medesimo genitore. Questa strategia ha la potenzialità di aumentare notevolmente il throughput a spese di un maggiore overhead sulle singole operazioni.

In un albero di **Phaser** stratificati, la registrazione e la cancellazione di un **Phaser** child con il rispettivo genitore avviene automaticamente. Ogni volta che il numero di soggetti iscritti ad un **Phaser** child diventa non-zero, il **Phaser** child viene automaticamente registrato con il relativo genitore. Analogamente, ogni volta che il numero di soggetti iscritti diventa pari a zero, come il risultato di una chiamata di **arriveAndDeregister**(), il **Phaser** child è cancellato dal suo genitore.

Monitoring

Mentre i metodi di sincronizzazione possono essere invocati soltanto dai soggetti registrati, tutti possono monitorare lo stato del **Phaser.** Questa feature è stata evidenziata nel codice seguente è stato creato appositamente un Thread con funzioni d Monitoring. I metodi disegnati appositamente per il monitoring sono **getRegisteredParties**() che permette di ottenere il numero dei soggetti registrati, **getArrivedParties**() che permette di ottenere il numero dei soggetti arrivati alla fase corrente (questo metodo è dotato del simmetrico **getUnarrivedParties**()), **getPhase**() restituisce il numero di fase corrente. Oltre a questi è sempre

possibile ottenere una snapshot dello stato del **Phaser** per mezzo dell'intramontabile metodo **toString**(). Da notare che **trattandosi di un sistema concorrente** (**multi-threaded**), **i valori restituiti da questi metodi possono riflettere stati transitori**.

Vantaggi principali della classe Phaser

Ricapitolando, i principali vantaggi della classe **Phaser**, rispetto alle precedenti implementazioni, possono essere riassunti nei seguenti punti:

- possibilità di registrazione dinamica dei soggetti alle varie fase;
- meccanismo di terminazione;
- maggiore flessibilità nella (ri)definizione del metodo (**onAdvance**) da invocare prima dell'avanzamento della fase;
- possibilità di organizzazioni oggetti **Phaser** in strutture ad albero;
- è predisposto per funzionare con il framework fork/join. Per esempio, dall'analisi di figura 3 è possibile notare che la classe annidata QNode implementa l'interfaccia ManagedBlocker definita all'interno (annidata) della classe ForkJoinPool.

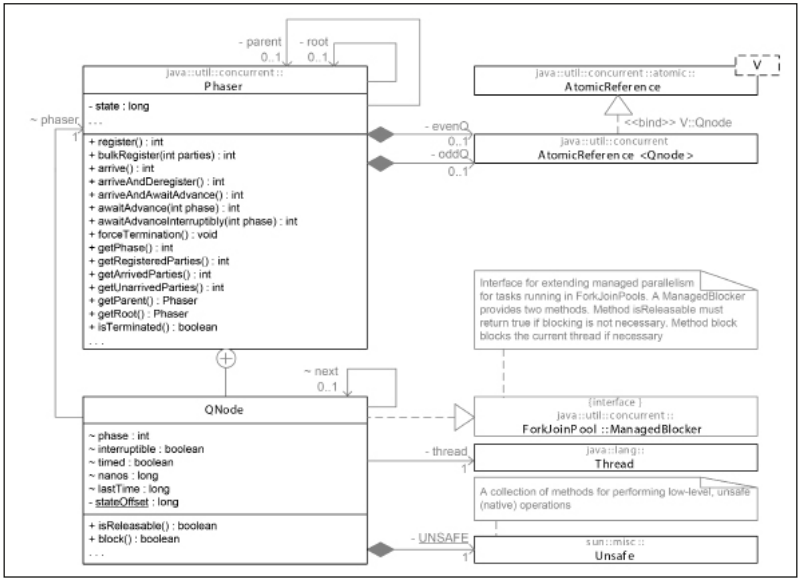


Figura 3. *Elementi principali della struttura della classe Phaser.*

Di seguito, riportiamo un esempio di utilizzo della classe Phaser.

```
public class PhaserExample {

  // ------------------- CONSTANTS SECTION -------------------
  /** logger */
  private static final Logger LOGGER =
        Logger.getLogger(PhaserExample.class);

  /** number of threads */
  private static final int NUM_THREADS = 4;
  // ------------------- ATTRIBUTES SECTION -------------------

  // ------------------- METHODS SECTION -------------------

  /**
   * Creates the required number of tasks.
   * Assigns each of them to a new thread and starts the thread
   */
  public static void createAndRunTasks() {

    Phaser phaser = new Phaser();

    // Monitor task
    MonitorPhaser aMonitorPhaser =
          new MonitorPhaser(phaser);
    new Thread(aMonitorPhaser).start();

    // Other tasks
    for (int i = 0; i < NUM_THREADS; i++) {
      Task aTask = new Task(phaser);
      new Thread(aTask).start();
    }
  }


  /**
   * @return the current thread id
```

```
 */
private static String getCurrentThreadId() {

  String threadId = Thread.currentThread().getId() + "";
  if (Thread.currentThread().getId() < 10) {
    threadId = "0" + threadId;
  }

  return threadId;
}

/**
 * Print out the current thread the given text and few data
 * about the given phaser status
 *
 * @param aPhaser   The phaser object
 * @param txt       Text to display
 */
private static void dumpStatus(Phaser aPhaser, String txt) {

  LOGGER.info(
     txt+"\t"+
     " Thread :" + getCurrentThreadId()+
     " Phase: " + aPhaser.getPhase() + " reg.:"+
     aPhaser.getRegisteredParties() + " arrived:"+
     aPhaser.getArrivedParties() + " unarrived:"+
     aPhaser.getUnarrivedParties());
}

/**
 * Inner class that defines the runnable task
 */
static class MonitorPhaser implements Runnable {

  private final Phaser aPhaser;
  public MonitorPhaser(Phaser givenPhaser) {
    aPhaser = givenPhaser;
  }
```

```
  @Override
  public void run() {

    while ( (aPhaser != null) &&
        (!aPhaser.isTerminated()) ) {

      PhaserExample.dumpStatus(
          aPhaser,
          "* * * MONITOR * * * ");

      try {
        Thread.sleep(80);
      } catch (InterruptedException e) {
        LOGGER.info("Ignore exception:" + e.getMessage());
      }
    }

    LOGGER.info("* * * MONITOR Shutting down* * * ");
  }
}

/**
 * Inner class that defines the runnable task
 */
static class Task implements Runnable {

  private final Phaser aPhaser;

  public Task(Phaser givenPhaser) {
    aPhaser = givenPhaser;
  }

  @Override
  public void run() {
    aPhaser.register();

    execNextPhase("Phase A");
```

```
    execNextPhase("Phase B");

    // at some point the task de-register itself from the phaser
    int newArrivalPhase =
        aPhaser.arriveAndDeregister();
    dumpStatus(
        aPhaser,
        "Task deregistered. Returned:"+ newArrivalPhase);
  }

  /**
   * Pretend to execute the work required by the next phase
   * @param phaseName  phase logical name
   */
  private void execNextPhase(String phaseName) {

    boolean last = false;
    pretendToDoSomeWork();

    String txt = " Completed phase.....";
    if (aPhaser.getUnarrivedParties() == 1) {
       txt =" Last thread arrived.";
       last = true;
    }
    dumpStatus(aPhaser, phaseName + txt);

    int arrivalPhase = aPhaser.arrive();

    // let's synch all thread
    dumpStatus(
        aPhaser,
        phaseName + " Await to advance....");
    aPhaser.awaitAdvance(arrivalPhase);
    dumpStatus(
        aPhaser,
        phaseName + " Advance next phase.");

    if (last) {
```

```
        LOGGER.info("---------------------------------- ");
      }
    }

    /**
     * Pretend to do some work...
     * But it just waits for a bit
     */
    public void pretendToDoSomeWork() {

      LOGGER.info(
          "Working.....................\t"+
          " Thread :" + getCurrentThreadId());

      try {
        Thread.sleep(
          ThreadLocalRandom.current().nextInt(10,400));
      } catch (InterruptedException e) {
        LOGGER.info("Ignore exception:" + e.getMessage());
      }
    }
  }

  /**
   * Poor way to test classes!
   *
   * @param args
   */
  public static void main(String[] args) {
    PhaserExample.createAndRunTasks();
  }
} // --- end of class ---
```

Listato 3. Esempio di utilizzo di Phaser.

Il metodo **main** invoca **createAndRunTasks**, che si occupa di creare un Thread che esegue il monitoraggio dell'oggetto Phaser ed una serie di oggetti **Task** (in

questo caso 4) ai quale viene fornito come parametro del costruttore la referenza al medesimo oggetto **Phaser**. Lo stesso metodo assegna ciascuna istanza **Task** ad un Thread dedicato che, una volta avviato (invocazione del metodo **start**), ne esegue il metodo run. Quest'ultimo esegue alcune semplici azioni, come registrarsi presso l'oggetto **Phaser** (**register**) e quindi eseguire due fasi: "Phase A" e "Phase B". Ciascun Thread, per ogni fase esegue i seguenti passi:

- esegue (o meglio pretende di eseguire, metodo **pretendToDoSomeWork**) il lavoro che precede la necessità di sincronizzarsi;
- segnala di essere arrivato alla fase (**in arrivalPhase = aPhaser.arrive**()).
- si pone in stato di attesa per la determinata fase (**aPhaser.awaitAdvance**(**arrivalPhase**)). In altre parole attende l'arrivo dei restanti Task registrati per la medesima fase per poter proseguire con il proprio lavoro.

Da notare che mentre **aPhaser.arrive**() è **un'operazione non bloccante**, **aPhaser.awaitAdvance**() **lo è**. Questo è evidente anche dall'output riportato di seguito: i vari Thread segnalano di aver completato la fase e poi di porsi in stato di attesa (testo: "Completed phase..." e " Await to advance..."). Poi però attendono prima di poter procedere alla fase successiva ("Advance next phase"). Per finire è stato creato un Thread di monitor che ciclicamente esegue l'output dello stato della barriera senza registrarsi ad alcuna fase. A questo punto il log riportato di seguito non dovrebbe presentare alcuna sorpresa. Da notare che per questioni di spazio i dati strettamente relativi al logging (classpath, timestamp, etc.) sono stati rimossi.

```
- dumpStatus:62 - * * * MONITOR * * *  Thread :08 Phase: 0 reg.:0
arrived:0 unarrived:0
- pretendToDoSomeWork:162  - Working.... Thread :09
- pretendToDoSomeWork:162  - Working.... Thread :11
- pretendToDoSomeWork:162  - Working.... Thread :10
- pretendToDoSomeWork:162  - Working.... Thread :12
```

Dall'analisi delle prime righe si nota: una prima esecuzione del Thread di monitoring che riporta uno stato praticamente non inizializzato dell'oggetto Phaser, l'avvio dei 4 Thread immediatamente "bloccati" nell'esecuzione del lavoro iniziale richiesto dalla fase A.

```
- dumpStatus:62  - Phase A Completed phase... Thread :09 Phase: 0
reg.:4 arrived:0 unarrived:4
- dumpStatus:62  - Phase A Await to advance.. Thread :09 Phase: 0
reg.:4 arrived:1 unarrived:3
```

```
- dumpStatus:62  - * * * MONITOR * * *        Thread :08 Phase: 0
reg.:4 arrived:1 unarrived:3
- dumpStatus:62  - Phase A Completed phase... Thread :11 Phase: 0
reg.:4 arrived:1 unarrived:3
- dumpStatus:62  - Phase A Await to advance.. Thread :11 Phase: 0
reg.:4 arrived:2 unarrived:2
- dumpStatus:62  - Phase A Completed phase... Thread :12 Phase: 0
reg.:4 arrived:2 unarrived:2
- dumpStatus:62  - Phase A Await to advance.. Thread :12 Phase: 0
reg.:4 arrived:3 unarrived:1
- dumpStatus:62  - * * * MONITOR * * *        Thread :08 Phase: 0
reg.:4 arrived:3 unarrived:1
- dumpStatus:62  - * * * MONITOR * * *        Thread :08 Phase: 0
reg.:4 arrived:3 unarrived:1
- dumpStatus:62  - * * * MONITOR * * *        Thread :08 Phase: 0
reg.:4 arrived:3 unarrived:1
- dumpStatus:62  - Phase A Last thread.       Thread :10 Phase: 0
reg.:4 arrived:3 unarrived:1
- dumpStatus:62  - Phase A Advance next phase.Thread :12 Phase: 1
reg.:4 arrived:0 unarrived:4
- pretendToDoSomeWork:162  - Working.........Thread :12
- dumpStatus:62  - Phase A Await to advance...Thread :10 Phase: 1
reg.:4 arrived:0 unarrived:4
- dumpStatus:62  - Phase A Advance next phase.Thread :10 Phase: 1
reg.:4 arrived:0 unarrived:4
- execNextPhase:152  - ----------------------------------
```

Il secondo segmento del log mostra che i Thread cominciano gradualmente a giungere alla prima sincronizzazione (phase 0). L'ultimo ad arrivare è il Thread 10 che conclude la sincronizzazione inziale e sblocca i Thread in attesa (riga evidenziata in neretto). Per evidenziare questo evento, viene stampate la linea di trattini orizzontali (l'ultima riga del frammento). Da notare che questo output è mostrato dopo che alcuni Thread cominciano già ad avanzare alla fase successiva. Questo perché l'output è parte di un programma multi-threading, pertanto, non deterministico. In questo caso è legittimo attendersi che il Thread 10 venga bloccato non appena giunge alla fase e sblocca gli altri Thread. Quando poi riprende richiede di mostrare a video la linee tratteggiata. Come si vede dalla porzione di log precedente, il Thread di monitoring di tanto in tanto entra in esecuzione ed

esegue il monitor dell'oggetto Phaser che in questo caso consiste nella semplice stampa di alcune sue informazioni.

```
- pretendToDoSomeWork:162  - Working.........Thread :10
- dumpStatus:62  - Phase A Advance next phase.Thread :11 Phase: 1
reg.:4 arrived:0 unarrived:4
- pretendToDoSomeWork:162  - Working.........Thread :11
- dumpStatus:62  - Phase A Advance next phase.Thread :09 Phase: 1
reg.:4 arrived:0 unarrived:4
- pretendToDoSomeWork:162  - Working.........Thread :09
- dumpStatus:62  - * * * MONITOR * * *        Thread :08 Phase: 1
reg.:4 arrived:0 unarrived:4
- dumpStatus:62  - Phase B Completed phase....Thread :10 Phase: 1
reg.:4 arrived:0 unarrived:4
- dumpStatus:62  - Phase B Await to advance...Thread :10 Phase: 1
reg.:4 arrived:1 unarrived:3
- dumpStatus:62  - * * * MONITOR * * *        Thread :08 Phase: 1
reg.:4 arrived:1 unarrived:3
- dumpStatus:62  - Phase B Completed phase... Thread :11 Phase: 1
reg.:4 arrived:1 unarrived:3
- dumpStatus:62  - Phase B Await to advance...Thread :11 Phase: 1
reg.:4 arrived:2 unarrived:2
- dumpStatus:62  - * * * MONITOR * * *        Thread :08 Phase: 1
reg.:4 arrived:2 unarrived:2
- dumpStatus:62  - Phase B Completed phase... Thread :09 Phase: 1
reg.:4 arrived:2 unarrived:2
- dumpStatus:62  - Phase B Await to advance...Thread :09 Phase: 1
reg.:4 arrived:3 unarrived:1
- dumpStatus:62  - * * * MONITOR * * *        Thread :08 Phase: 1
reg.:4 arrived:3 unarrived:1
- dumpStatus:62  - Phase B Last thread.       Thread :12 Phase: 1
reg.:4 arrived:3 unarrived:1
- dumpStatus:62  - Phase B Advance next phase.Thread :09 Phase: 2
reg.:4 arrived:0 unarrived:4
- dumpStatus:62  - Phase B Await to advance...Thread :12 Phase: 2
reg.:4 arrived:0 unarrived:4
- dumpStatus:62  - Phase B Advance next phase.Thread :11 Phase: 2
reg.:3 arrived:0 unarrived:3
```

```
- dumpStatus:62  - Task deregistered. Ret.:2  Thread :11 Phase: 2
reg.:2 arrived:0 unarrived:2
- dumpStatus:62  - Task deregistered. Ret.:2  Thread :09 Phase: 2
reg.:3 arrived:0 unarrived:3
- dumpStatus:62  - Phase B Advance next phase.Thread :10 Phase: 2
reg.:4 arrived:0 unarrived:4
- dumpStatus:62  - Task deregistered. Ret.:2  Thread :10 Phase: 2
reg.:1 arrived:0 unarrived:1
- dumpStatus:62  - Phase B Advance next phase.Thread :12 Phase: 2
reg.:3 arrived:0 unarrived:3
- execNextPhase:152  - ----------------------------------
- dumpStatus:62  - Task deregistered. Ret.:2  Thread :12 Phase:
-2147483645 reg.:0 arrived:0 unarrived:0
- run:97  - * * * MONITOR Shutting down* * *
```

La parte finale del log ripete i concetti illustrati in precedenza, unica differenza è la parte finale. In particolare, una volta che tutti i Thread sono stati sbloccati dalla fase 2, l'ultima "deregistrazione" restituisce come valore di ritorno un numero negativo (penultima riga) proprio a testimoniare che il Phaser è in fase di terminazione. Anche il Thread di monitoring riceve questa informazione è quindi esegue la propria chiusura.

Il framework fork-join

Il **framework fork-join è indubbiamente la feature più interessante** introdotta nel package della concorrenza con Java SE 7 il cui core è la classe **ForkJoinPool,** che specializza la classe astratta **AbstractExecutorService** per l'esecuzione di task di tipo **ForkJoinTask** (cfr. figura 4).

Questi ultimi sono istanze della classe astratta **ForkJoinTask<V>** che implementa l'immancabile interfaccia **Serializable** e l'interfaccia **Future<V>** che rappresenta **una sorta di "Thread"** dal punto di vista della **possibilità di eseguire task asincroni possibilmente in parallelo**. La classe ForkJoinWorkerThread, specializzazione della classe Thread, si occupa di eseguire i lavori assegnati al pool (gestisce una coda di istanze che specializzano ForkJoinTask: ForkJoinTask<?>[] queue).

Da notare che il framework utilizza una strategia più leggera rispetto all'assegnazione di un lavoro ad un Thread: attraverso l'idea di pool (ForkJoinWorkerThread[] workers) il framework è in grado di assegnare un gran numero di task e sub-task ad un ridotto numero di Thread.

Un oggetto **ForkJoinTask** inizia la propria esecuzione dopo essere stato sottoposto a un **ForkJoinPool** (invocazione di uno dei metodi **execute**, **invoke**, **submit**, etc. come dettagliato nella tabella 1). Una volta avviato, il comportamento tipico prevede la creazione a sua volta di sotto attività (comportamento tipico degli algoritmi **divide and conquer**, figura 5). Come indicato dallo stesso nome di questa classe, molti programmi che utilizzano **ForkJoinTask** ricorrono all'utilizzo dei soli metodi di **fork**() e **join**() (o derivati come **invokeAll**). Questa classe fornisce anche altri metodi molto utili per utilizzi avanzati e richiesti per l'inserimento nella gerarchia dei Future (gli oggetti **ForkJoinTask** sono versioni "leggere" di **Future**).

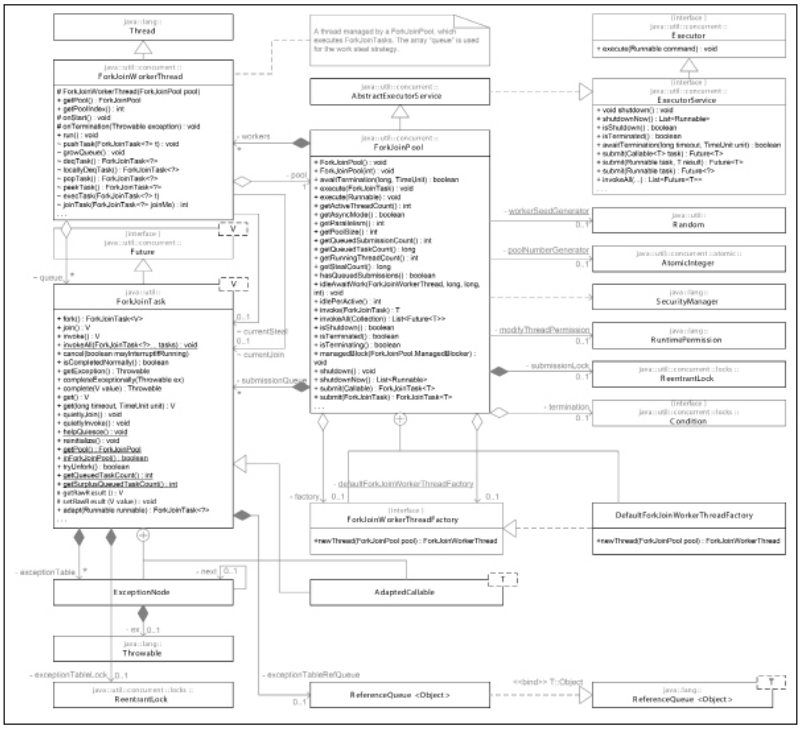


Figura 4. *Class diagram del nuovo framework fork/join. Nota bene: la quasi totalità delle classi mostrate in figura 4 che formano il framework fork/join fa ricorso a codice nativo per mezzo del solita classe* **Unsafe**. *Tuttavia tale classe non è stata inserita nel diagramma esclusivamente per questioni di spazio.*

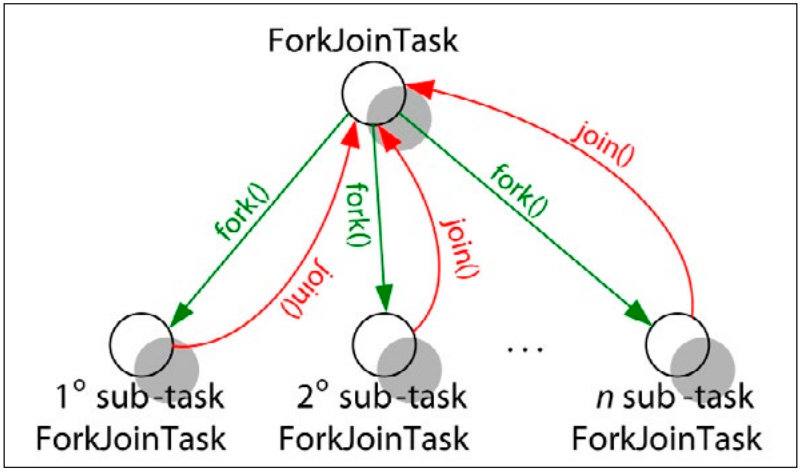


Figura 5. *Comportamento tipico di un algoritmo fork/join.*

L'efficienza delle istanze **ForkJoinTask** è ottenuta anche per mezzo di una serie di restrizioni (spesso informali e quindi solo parzialmente applicabili staticamente) che ne riflettono la destinazione d'uso, come per esempio utilizzo per compiti computazionali di puro calcolo ed esecuzione di specifici algoritmi su oggetti isolati (non condivisi). **I meccanismi primari di coordinamento** sono dati dal metodo **fork**(), che **organizza l'esecuzione asincrona**, e **join**(), che **blocca l'esecuzione fino al termine** dell'elaborazione del risultato dell'attività (cfr. figura 5). Il comportamento richiesto prevede che i calcoli nei vari oggetti **non necessitino di sincronizzazioni/blocchi e comunicazioni esplicite** oltre quelle richieste dal FFJ. Inoltre i vari task non dovrebbero eseguire operazioni di I/O bloccanti, e dovrebbero accedere (idealmente) alle sole variabili che sono del tutto indipendenti da quelle accessibili da altri processi in esecuzione. Come è lecito attendersi, violazioni minori di queste restrizioni non sono un problema. Per esempio, non è grave far sì che diversi task accedano contemporaneamente in sola lettura ad una medesima struttura dati o che utilizzino stream in uscita condivisi. Tuttavia è importante tener presente che **il FFJ non è stato pensato per utilizzi che richiedono continuo coordinamento dei vari Thread**, continui accessi contesi, e simili. che riducono le prestazioni e possono generare potenziali situazioni di stallo a tempo indeterminato. Questa restrizione di utilizzo può essere solo parzialmente forzata. Una strategia utilizzata per

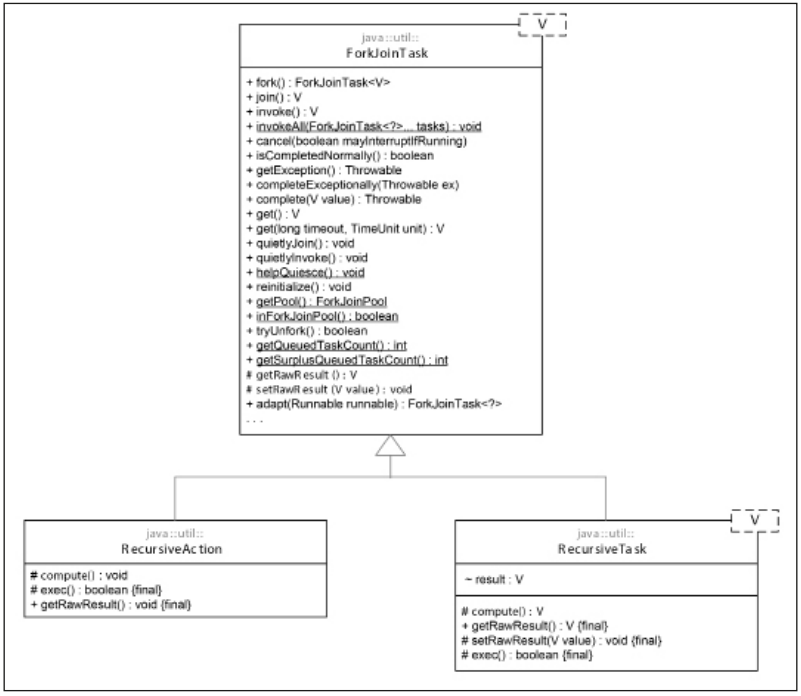


Figura 6. *Le sotto-classi di ForkJoinTask: RecursiveAction e RecursiveTask.*

forzare questa limitazione consiste nel non consentire la generazione di eccezioni checked come **IOException**. Tuttavia, i task possono sempre generare eccezioni run-time, come **RejectedExecutionException** che viene scatenata qualora si assista ad un esaurimento delle risorse interne, come per esempio il fallimento dell'allocazione delle code dei task interni.

La classe ForkJoinTask raramente è utilizzata direttamente (eseguirne una specializzazione non è una pratica consigliata). La strategia decisamente più frequente e raccomandata è basata sull'implementazione di un'apposita sottoclasse di una delle sue classi astratte (cfr. figura 6).

RecursiveAction: da utilizzarsi per l'esecuzione di algoritmi che non restituiscono risultati. Per questo motivo RecursiveAction non è una classe tipizzata. La relativa dichiarazione prevede:

```
public abstract class RecursiveAction extends ForkJoinTask<Void>
```

RecursiveTask: da utilizzarsi per calcoli che generano un risultato (caso più frequente). Questo è memorizzato nell'attributo result e viene impostato e restituito dai metodi predefiniti e final: **getRawResult** e **setRawResult**.

Il furto del lavoro

Il funzionamento del **FFJ** è una variante dello scheduler (basato sul **principio del work stealing**, "furto del lavoro") del linguaggio di programmazione Cilk (cfr. [Leiserson, 2010]): si tratta di un linguaggio di programmazione e di un ambiente run-time, studiato presso il MIT, disegnato appositamente per l'implementazione di algoritmi parallelizzabili. La filosofia base del linguaggio è che **un programmatore dovrebbe concentrarsi nello strutturare il proprio programma al fine di esporre il parallelismo e sfruttare località, lasciando al sistema run-time la responsabilità di schedulare il calcolo** da eseguire in modo efficiente sulla piattaforma data. Il sistema Cilk run-time si prende cura dei dettagli come protocolli di bilanciamento del carico, la sincronizzazione e la comunicazione. Il principio di funzionamento del FFJ è basato sull'organizzazione descritta di seguito.

Contesto di invocazione / Tipo di esecuzione	*Invocazione "esterna" (non da un task del pool)*	*Invocazione "interna" (da un task fork/join in esecuzione)*
Asincrona	execute(ForkJoinTask)	ForkJoinTask.fork()
Bloccante (attesa dell'ottenimento del risultato)	invoke(ForkJoinTask)	ForkJoinTask.invoke()
Richiede esecuzione con l'ottenimento di un oggetto Future	submit(ForkJoinTask)	ForkJoinTask.fork() (ForkJoinTask sono Future)

Tabella 1. *Tipi di invocazione ad un'istanza ForkJoinPool.*

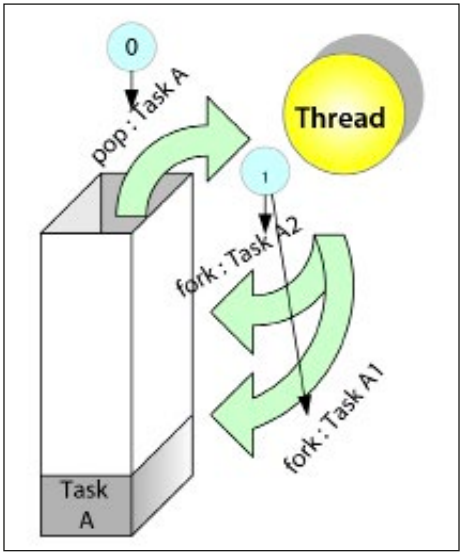


Figura 7. *Una rappresentazione grafica della strategia work-stealing.*

Ogni Thread (ForkJoinWorkerThread) mantiene i propri task da eseguire in una coda di pianificazione lavori dedicata (attributo queue).

Le code sono di tipo double-ended (interfaccia **Deque** estendente **Queue**), chiamate così perché supportano sia una semantica tipica delle code FIFO (first in first out, primo dentro, primo fuori), sia degli stack LIFO (Last In First Out, l'ultimo dentro è il primo fuori). Pertanto hanno due vie di uscita che ne caratterizzano il nome.

Le sotto-attività generate dalle attività gestite da un determinato thread sono, tipicamente, inserite nella code double-ended dello stesso thread.

Ogni thread esegue i task nella propria coda attraverso una strategia a pila: il task più giovane è eseguito prima (il thread preleva i lavori da eseguire invocando un'operazione pop sulla coda). Quando un thread non ha più attività da eseguire nella propria coda va in cerca di attività da eseguire ("rubare") nella coda di un Thread del pool scelto casualmente. In questo caso **tenta di prelevare il task che da più tempo è nella coda** (regola FIFO).

Quando un Thread incontra un'operazione di join, continua a elaborare gli altri task (se disponibili), fino a quando il task di destinazione è stato completato (il

relativo metodo **isDone** restituisce il valore **true**). A parte l'operazione di join, tutti i task proseguono fino al completamento senza blocchi. Quando poi un thread ha esaurito tutti i task dalla propria coda e non riesce a rubare un task da un'altra coda, si defila (esegue un'operazione, **yield**, **sleep**, etc., che lo forza a lasciare lo stato di esecuzione), per riprovare in un secondo momento. Ciò a meno che tutti i thread si trovino in un medesimo stato di "idle". In qual caso, tutti i thread vanno in uno stato di idle finché non viene richiesto un nuovo task è al livello più altro.

La figura 7 mostra un Thread che esegue i task presenti nella propria coda (Deque). In particolare, l'immagine mostra che il Thread preleva un task dalla propria coda (pop) e quindi, eseguendo un tipico algoritmo Diviti & Conquer (D&C, diviti et impera), genera nuovi task (sub-task) che vengono inseriti nella stessa coda (push).

La figura 8 mostra due Thread appartenenti al medesimo pool. Mentre Thread 2 ha molti Task da eseguire, il Thread 1 non ne ha. Quindi, grazie al supporto del framework, ruba il lavoro più anziano presente nella coda del Thread 2 per eseguirlo. A tal fine vengono utilizzate le feature tipiche della coda rispetto a quelle della pila, al fine di rubare il lavoro con maggiore anzianità. L'utilizzo di diverse

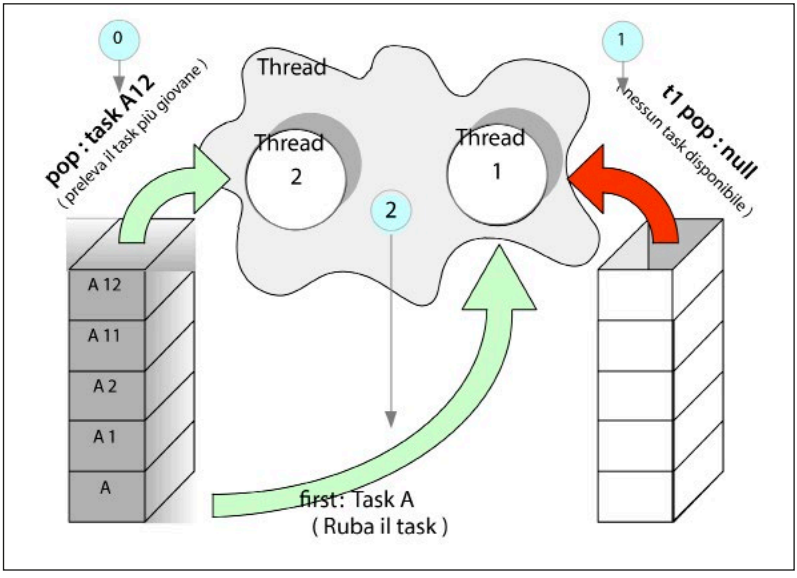


Figura 8. *Un'ulteriore rappresentazione grafica della strategia work-stealing.*

strategie di gestione delle stessa "coda" permette di evitare "conflitti" (o comunque di minimizzarne l'occorrenza) con il Thread 2.

La tecnica del work stealing permette di creare scheduler leggeri ed estremamente efficienti che grazie all'oculata strategia di allocazione e prelevamento dei task permette, costruttivamente, di minimizzare contese. A questo punto dovrebbe essere chiaro il perché è opportuno realizzare task che non necessitano ulteriori sincronizzazioni a parte quelle proprie del FFJ, proprio per non collidere con la politica di gestione dello stesso framework.

Critiche al FFJ

Per questioni di completezza, diciamo anche che in Internet è possibile reperire alcuni articoli, non molti a dire il vero, che esprimono pareri molto critici rispetto al FFJ considerato un mero esercizio accademico con ridotte possibilità di utilizzo concreto in ambienti professionali. L'accusa principale è relativa al fatto che l'implementazione di sistemi complessi richiede soluzioni MT e la parallelizzazione di processi che vanno ben oltre al pattern D&C e al coordinamento di singole classi/Thread. I vari articoli, a onor del vero, hanno sicuramente qualche spunto interessante: per esempio Edward Harned (cfr. [Harned, 2012]), critica **anche l'eccessivo utilizzo di sun.misc.Unsafe**, ossia il collegamento a routine scritte in C dipendenti dalla piattaforma al fine di ottimizzare le performance. Ma sul resto, crediamo che si tratti più di una provocazione che di un'argomentazione solida e che le soluzioni vadano valutate nel rispettivo contesto.

Per esempio, lo stesso Lea non ha mai posizionato il FFJ come backbone di una soluzione Enterprise. Non è mai stato venduto come un'alternativa agli application server, sebbene sia possibile criticare il fatto che l'utilizzo del FFJ nel contesto degli application server dovrebbe essere considerato accuratamente dal momento che **è compito dell'EJB container di gestire Thread** e le **implementazione dei componenti non dovrebbero, direttamente o indirettamente, lanciare nuovi Thread**. il FFJ è stato concepito come una soluzione efficiente ed elegante per supportare l'implementazione di algoritmi, come per esempio la ricerca del best-path, che possono trarre grandi benefici da una struttura D&C.

Invece uno spunto interessante è quello di vedere quale sarà l'utilizzo del FFJ ed il relativo sviluppo quando verrà finalmente rilasciata la grammatica delle **espressioni Lambda**, che nativamente, dovrebbe includere particelle grammaticali che consentono allo sviluppatore di indicare l'esecuzione parallela (parallel()) di determinate funzioni come riportato di seguito.

```
int sum = dvds.parallel()
    .filter(b -> b.getAuthor() == "Vaya con Dios")
    .map(b -> b.getTracks())
    .sum();
```

Probabilmente il framework che si incaricherà di eseguire questo codice utilizzerà dietro le quinte proprio il framework fork and join.

Algoritmi "Divide et Impera" (Divide and Conquer, D&C)

Il FFJ è ideale per l'implementazione di algoritmi D&C la cui struttura tipica prevede un pattern come quello descritto di seguito:

```
0. Risultato risolviProblema(Problema problema) {
1.   if (il problema è sufficientemente piccolo) {
2.     risolvi il problema
3.   } else {
4.     Decomponi il problema in sotto-problemi indipendenti
5.     (fork) genera sotto-task per risolvere i sotto-problemi
6.     (join) riunisci i sotto-task
7.     componi il risultato a partire dai quelli parziali
     generati dai sotto-task
8.   }
9. }
```

Listato 4. *Struttura tipica di un algoritmo D&C.*

Da una prima analisi del precedente listato si può notare che si tratta di una struttura ricorsiva nella quale è possibile individuare:

- **il passo base**: (linee 1 e 2) che permette di terminare le invocazioni ricorsive;
- **il passo ricorsivo**: (linee da 4 a 7), che si occupano di decomporre il problema dato in problemi più semplici da risolvere indipendentemente dallo stesso algoritmo per poi ricomporre il risultato.

Qualche nota storica

L'algoritmo D&C più popolare, che verosimilmente tutti i lettori avranno implementato almeno una volta, è la ricerca binaria. Una descrizione dettagliata si deve a John Mauchly (fisico americano che con J. Presper Eckert disegnò l'ENIAC) che nel 1946 dettagliò il modello di ricerca binaria. Tuttavia tracce

di algoritmi simili risalgono addirittura al 200 a.C. ([Knuth, 1998]). L'idea alla base è di partire da un array di elementi ordinati e quindi eseguire la ricerca dividendo ogni volta l'array oggetto di ricerca in due componenti: quelli degli **elementi inferiori** alla chiave da cercare e quelli degli **elementi superiori** alla chiave che come tale possono essere ignorati. Altro algoritmo D&C, che risale a diversi secoli prima di Cristo, è **l'algoritmo di Euclide** atto a calcolare il massimo comun divisore di due numeri la cui strategia consiste nel ridurre i numeri in equivalenti sotto-problemi sempre più piccoli. Inoltre è possibile menzionare **la serie di Fibonacci** che si presta egregiamente ad un'implementazione D&C.

Un primo esempio di algoritmo D&C con sotto-problemi multipli è **l'algoritmo di Gauss** coniato nel 1805 divenuto poi famoso con il nome di **trasformata veloce di Fourier** (**FFT**, Fast Fourier Transformer) di Cooley-Tukey che lo riscoprirono circa un secolo più tardi. Mentre, uno dei primi algoritmi D&C **specificatamente sviluppato per i computer** è l'algoritmo di ordinamento **merge sort** inventato da John von Neumann nel 1945 del quale riportiamo l'implementazione fork/join di seguito.

Altro celebre esempio di algoritmo D&C che, inizialmente, non prevedeva un aspetto di automatizzazione è il **metodo dell'ufficio postale** coniato da Knuth. Si tratta di una strategia utilizza per organizzare razionalmente l'instradazione della posta. La strategia prevede che le lettere siano ordinate in sacchi separati per le diverse aree geografiche. Ognuno di questi sacchi di lettere viene poi suddiviso in lotti per aree più piccole, e così via fino alla consegna. Si tratta di una strategia simile al radix-sort implemetato attraverso schede perforate nel 1929.

Merge Sort

Il seguente listato riporta lo **pseudo-code** dell'algoritmo **merge sort**.

```
 function mergeSort(List aList)
   // se la lisa ha un solo elemento, è ordinata. Ritorna la
lista
   if aList.size() <= 1
     return aList
   //...altrimenti, suddividi la lista in due sotto liste
   List leftList
   List rightList
   int middle = aList.size() / 2
   for each element in aList before middle
     leftList.add(element)
```

```
    for each element in aList after or equal middle
      rightList.add(element)
    // invoca ricorsivamente mergeSort.
    // ciò comporta l'ulteriore suddivisione delle sotto liste
    // fino al passo base (lista unitaria)
    leftList = mergeSort(leftList)
    rightList = mergeSort(rightList)
    // esegui la fusione delle sottoliste restituite dalle
    // precedenti invocazioni mergeSort in una nuova lista
    // restituita al chiamante
    return merge(left, right)
```

Listato 5. *Pseudo-codice dell'algoritmo merge-sort.*

Ed ecco di seguito il listato con l'implementazione dell'algoritmo MergeSort eseguita con il FFJ:

```
/**
 * This class implements the Quick Sort algorithm
 * The complexity is mostly O(n Log n), in worst case scenarios
can be O(n^2)
 * Algorithm:
 * 1. pick an element from the list called the pivot
 * 2. Partition the list:
 *   - all elements with value less than the pivot come before the
pivot
 *   - all elements with value greater than the pivot come after
the pivot
 *   - elements with the same value can go either way as long as
it is consistent
 * 3. Recursively
 *      - sort sub-list with elements prior the pivot
 *      - sort sub-list with elements after the pivot
 * 4. Terminate the execution when the list has 1 or zero elements
 */

public class   QuickSort<T extends Comparable<T>>
```

```
    extends RecursiveAction {


  // --------------------- CONSTANTS SECTION ---------------------
  /** default serial version ID */
  private static final long serialVersionUID = 1L;


  // --------------------- ATTRIBUTES SECTION -------------------
  /** data source to sort */
  private List<T> dataSource;
  /** lower index where to start the sort. Used in recursive
invocations */
  private int     lowerIndex;
  /** upper index where to stop the sort. Used in recursive
invocations */
  private int     upperIndex;
  /** kind of unique task id */
  private long    taskId;

  // --------------------- METHODS SECTION ---------------------

  // ------------------------------------------- constructors
  /**
   * Full constructor
   * @param dataSource
   */
  public QuickSort(List<T> dataSource) {
    this.dataSource = dataSource;
    this.lowerIndex = 0;
    this.upperIndex = dataSource.size() - 1;
    this.taskId =  System.nanoTime();

    outputStat("New (first)  ");
  }

  /**
   * Full constructor
```

```
     * @param dataSource
     * @param lowerIndex
     * @param upperIndex
     */
    protected QuickSort(
            List<T> dataSource,
            int lowerIndex,
            int upperIndex,
            long taskId) {

      this.lowerIndex = lowerIndex;
      this.upperIndex = upperIndex;
      this.dataSource = dataSource;
      this.taskId     = taskId;

      outputStat("New          ");
    }


    // ---------------------------------------- specific methods

    @Override
    protected void compute() {

      if (lowerIndex < upperIndex) {
        outputStat("Compute Start");

        // middle point
        int pivotIndex =
           lowerIndex + ((upperIndex - lowerIndex) / 2);

        pivotIndex = partition(pivotIndex);

        invokeAll(
          new QuickSort<>(dataSource, lowerIndex,
                    pivotIndex-1, System.nanoTime()),
                    new QuickSort<>(dataSource, pivotIndex+1,
                    upperIndex, System.nanoTime())
```

```
    );
    outputStat("Compute End  ");
  }
}

/**
 * This is a simple output used to show the stealing strategy
 * @param operation   operation to print
 */
private void outputStat(String operation) {
  String threadId = Thread.currentThread().getId()+"";
  if (Thread.currentThread().getId() < 10) {
    threadId = "0"+threadId;
  }

  System.out.println(
        " task id:"+this.taskId+
        " thread:"+threadId+" op:"+operation);
}

/**
 * partition the data set
 */
private int partition(int pivotIndex) {
  T pivotValue = dataSource.get(pivotIndex);   // middle element
  swap(pivotIndex, upperIndex);

  int storeIndex = lowerIndex;
  // go through all elements
  for (int i=lowerIndex; i < upperIndex; i++) {

    if (dataSource.get(i).compareTo(pivotValue) < 0) {
      swap(i, storeIndex);
      storeIndex++;
    }
  }

  swap(storeIndex, upperIndex);
```

```
    return storeIndex;
  }

  /**
   * Swap the elements in the given positions
   */
  private void swap(int i, int j) {
    if (i != j) {
      T iValue = dataSource.get(i);

      dataSource.set(i, dataSource.get(j));
      dataSource.set(j, iValue);
    }
  }

  // -------------- MAIN METHOD: a poor way to test classes!!!
  public static void main(String[] args) {
    int MAX_SIZE = 10_000;

    // ------------------------- generate a random list
    List<Integer> sourceData = new ArrayList<>(MAX_SIZE);

    for (int i=0; i<MAX_SIZE; i++) {
      int randomValue = (int) (Math.random() * 100_000);
      sourceData.add(randomValue);
    }

    // ------------------------- execute the sort
    QuickSort<Integer> quickSort =
              new QuickSort<>(sourceData);

    ForkJoinPool pool = new ForkJoinPool(6);
    pool.invoke(quickSort);

  }
} // --- end of class --
```

Listato 6. *Implementazione dell'algoritmo Merge-sort per mezzo del framework ForkJoin.*

L'esecuzione di questo programma permette di evidenziare alcune interessanti caratteristiche.

In primo luogo viene utilizzato un limitato pool di Thread per eseguire la miriade di task generati durante l'esecuzione. In particolare, l'implementazione richiede un pool di 6 thread (**new ForkJoinPool(6)**). Ciò chiarisce il perché i **ForkJoinTask** siano considerati Thread "leggeri": il fork di un nuovo task non genera assolutamente la creazione di un nuovo thread e tanto meno quello di un nuovo processo.

I nuovi task possono venire allocati (dal gestore del pool) alla coda di un Thread diverso da quello che lo ha creato. I "thread" rubano spesso il lavoro dalla coda degli altri thread.

In merito all'ultimo punto, è possibile notare sequenze come le due riportate di seguito:

```
task id:4781408082098 thread:08 op:New
task id:4781408082098 thread:09 op:Compute End
task id:4781408082098 thread:09 op:Compute Start

task id:5762273738806 thread:08 op:New
task id:5762273738806 thread:11 op:Compute End
task id:5762273738806 thread:11 op:Compute Start
```

Nel primo caso il Thread 8 genera un nuovo task che poi viene acquisto dal Thread 9. La stessa sequenza accade successivamente, in questo caso il Thread "ladro" è il numero 11.

Sempre dall'analisi del codice è possibile notare che il disegno tipico di una specializzazione della classe astratta **ForkJoinTask** (ossia del codice che poi eseguirà il lavoro) richiede che le informazioni su cui dovrà operare siano fornite come parametri del costruttore, e che la parte di calcolo vera e propria utilizzi i metodi di controllo forniti da questa classe base: fork e join, oppure un metodo a più alto livello di astrazione, come **invokeAll** dell'esempio, che esegue il fork automatico di tutti i task definiti per poi eseguirne il join.

Per finire è possibile notare che nel codice si sono utilizzate alcune feature implementate dal progetto Coin, Java SE 7 descritte nell'articolo precedente, come, per esempio, l'utilizzo dell'underscore per aumentare il grado di leggibilità di alcuce costanti:

```
int MAX_SIZE = 10_000; e (int) (Math.random() * 100_000);
```

o il ricorso alla notazione contratta del diamante

```
= new QuickSort<>(sourceData);
```

e

```
QuickSort<>(dataSource, lowerIndex, pivotIndex-1, System.
nanoTime()),
```

Conclusioni

Architetture multi-core/multi-processore hanno ormai invaso il mercato di largo consumo. Diviene sempre più pressante avere software in grado di sfruttare al massimo le potenzialità di queste architetture. Nel mondo Java ciò si traduce in implementare software che la macchina virtuale (JVM, Java Virtual Machine), in collaborazione con i servizi del sistema operativo, può facilmente distribuire sui vari core/processori disponibili.

In questo capitolo sono state presentate le nuove feature del package della concorrenza introdotte con Java SE 7. In questo caso il giudizio non può che essere positivo... come del resto succede per la stragrande maggioranza dei lavori di Doug Lea. Le nuove feature continuano il percorso di revisione dell'iniziale strategia Java per il supporto al MT iniziato con J2SE 5.0. Sebbene la strategia iniziale alla base di Java fu di inserire il supporto per il MT nativamente nel linguaggio e, almeno inizialmente sembrava veramente un'ottima idea in discontinuità con i linguaggi del passato, ben presto tutte le sue limitazioni sono divenute evidenti: eccessiva rigidità (singolo monitor), generazione di colli di bottiglia (i meccanismi di controllo della concorrenza erano eccessivamente coarse-grained), feature troppo a basso livello (**Thread**, **synchronized**, **wait** e **notify**), ridotto controllo, etc. Ora, con il senno di poi è possibile affermare che tale supporto fu più una limitazione che un reale vantaggio.

Tra le varie feature presentate in questo capitolo, il framework Fork/Join è decisamente quella più interessante/importante. In particolare, introduce una semplice ed efficace tecnica di disegno per ottenere elevate prestazioni da algoritmi in grado di essere scomposti in sotto-parti in gradi di funzionare in parallelo (algoritmi Divide and Conquer). La sincronizzazione tra i vari Thread è costruttivamente ridotta al minimo grazie alla strategia utilizzata per far funzionare l'assegnazione dei vari lavori: ogni Thread utilizza la propria coda come una pila e quindi inserisce e preleva i lavori da eseguire sempre "sopra", mentre il framework può rimuovere i task da un Thread per assegnarli ad altri dal fondo: preleva i più

anziani secondo i dettami propri delle code. Il framework fork-join, se utilizzato secondo le direttive (non si creano task che richiedono ulteriori sincronizzazioni, continui accessi I/O, etc.) non blocca mai i vari Thread se non per attendere che tutte le varie sottoattività siano pronte quando è richiesto di esegurie il join. L'overhead e la sincronizzazioni sono quindi ridotto al minimo. Il framework fork/join mostra in maniera chiara come le migliori soluzioni frequentemente partono proprio dalla semplicità e pulizia del relativo disegno.

Capitolo 4
NIO 2

Introduzione

Una parte importante di Java SE 7 è indubbiamente rappresentata dalle innovazioni apportate alla API **NIO**: si tratta di una serie di variazioni abbastanza significative che hanno portato alla formulazione della versione NIO2. Le principali innovazioni sono condensate nei nuovi package: **java.nio.file**, **java.nio.file.attribute**, **java.nio.file.spi** e **com.sun.nio.sctp** (quest'ultimo contiene gli aggiornamenti necessari per supportare il nuovo protocollo **SCTP** descritto di seguito). Gli obiettivi primari di questa release (oltre a migliorie varie) puntano ad aumentare l'indipendenza dalla piattaforma di esecuzione, a fornire ulteriori servizi di alto livello e ad aggiungere il supporto per metadati e collegamenti simbolici come illustrato di seguito. Da notare che l'indipendenza dalla piattaforma è stata ottenuta (quasi paradossalmente) nuovamente sfruttando massicciamente i servizi offerti dal sistema operativo e da implementazioni proprietarie.

Una feature molto interessante, che condensa le varie innovazioni, è il nuovo meccanismo di **gestione dei file**. Sebbene la tradizionale classe **java.io.File** sia ancora disponibile, è stato introdotto un nuovo meccanismo che ha al centro le nuove classi: **java.nio.Path** e **java.nio.Files**. Queste offrono una serie di funzionalità molto potenti e un codice molto robusto. Inoltre sono stati aggiunti importanti **servizi di alto livello** come la copia ed il trasferimento di file e directory. Per quanto concerne il supporto per i metadati, è stato introdotto il package **java.nio.file.attribute** allo scopo di fornire un supporto superiore ai metadata dei file, quali il proprietario (owner, metodo **getOwner**), informazioni circa la natura del file stesso se si tratta di un link simbolico o di un file regolare (**isSymbolicLink**, **isRegularFile**), il reperimento del timestamp dell'ultima modifica (**getLastModifiedTime**), i permessi di un file (**getPosixFilePermissions**), e così via.

Java NIO2 inoltre fornisce un supporto esteso ai link simbolici. Anche se ciò era parzialmente possibile con le versioni precedenti, vi erano alcuni problemi: per esempio non era immediato, e spesso neanche completamente possibile, scrivere del codice robusto per la navigazione ricorsiva e la gestione dei link che generavano cicli di link simbolici.

Anche la **scalabilità** è stata attentamente rivista. In particolare, una serie di metodi di **java.io.file** non presentavano una scalabilità ottimale. Per esempio, la richiesta della lista di una directory di grandi dimensioni di un server poteva bloccare seriamente il programma, senza menzionare il fatto che poteva anche causare problemi con la memoria e quindi generare il rifiuto dell'erogazione del servizio. Il nuovo package di I/O è stato disegnato prendendo nella giusta considerazione questi elementi e per assicurarsi che i vari servizi funzionino come atteso, indipendentemente dalle dimensioni dei file e directory. Oltre a queste feature, si è fatto in modo che i vari servizi presentino lo stesso funzionamento in diverse piattaforme (in passato questo non era il caso per una serie di servizi come per **java.io.File.renameTo**).

Per alcune informazioni circa l'API NIO si rimanda all'apposita appendice inclusa in questo libro.

java.nio.file.FileSystems e FileSystem

La nuova versione **NIO2** introduce tutta una serie di funzionalità tipicamente fornite dalla piattaforma sottostante (sistema operativo e, più precisamente, file system) di cui la libreria fa largo utilizzo. Tuttavia, in alcuni casi in cui le funzionalità richieste non siano disponibili, è prevista un'**implementazione nativa a basso livello** (per esempio è il caso del **watch service**). Come di consueto, il

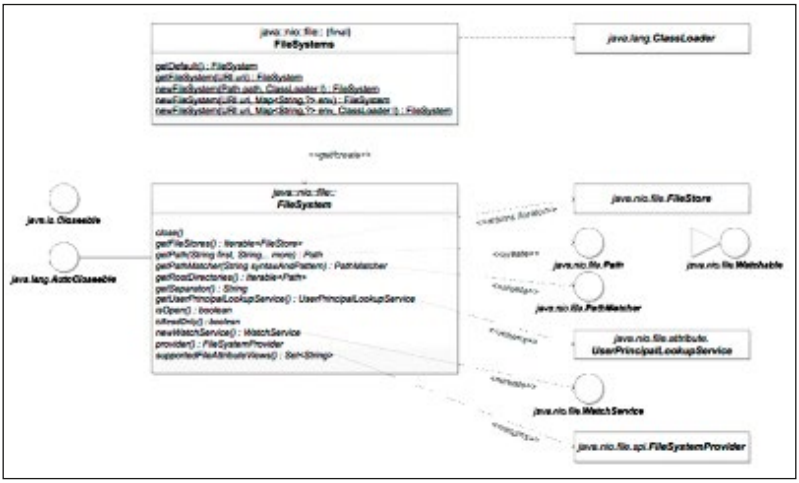


Figura 1. *Diagramma delle classi di FileSystems e FileSystem.*

modello implementato sfrutta la classica strategia Java: si definisce un framework che dipende da una serie di interfacce e classi astratte (figura 1) le cui implementazioni sono fornite da appositi **service provider** (ciò che una volta venivano chiamati **driver**). Nel caso in questione, i vari servizi sono forniti da implementazioni della classe astratta **FileSystemProvider**. La sostituzione di implementazioni concrete avviene attraverso apposite classi **factory** utilizzate sia per individuare le implementazioni delle specifiche interfacce e classi astratte (è il caso della classe **FileSystem**), sia per individuare lo specifico service provider (è il compito della classe **FileSystems**). Questo meccanismo è evidenziato dal diagramma delle classi di Figura 1.

Funzionamento

La classe **(final) FileSystems** contiene una serie di metodi factory sia per reperire l'oggetto **java.nio.file.FileSystem** di default, sia per generarne nuovi. La prima invocazione di uno dei metodi forniti da **FileSystems** genera il caricamento e l'inizializzazione del provider di default (secondo una classica strategia **lazy loading**), il quale, identificato dalla URI schema "file", si occupa di creare l'oggetto **FileSystem** che fornisce l'accesso alla lista di file system accessibili dalla JVM. Se i processi di caricamento o di inizializzazione del provider predefinito falliscono, viene generato un errore non specificato.

Da notare che la "risorsa" **FileSystem** non può essere chiusa, né esplicitamente mediante chiamata dell'apposito metodo **close**, né implicitamente per mezzo del costrutto **try-with-resource**. Ogni tentativo di chiudere il **FileSystem** di default genera una **UnsupportedOperationException**.

La prima chiamata di uno dei metodi **newFileSystem** (sempre forniti dalla classe **FileSystems**) genera l'individuazione e il caricamento di tutti i provider di file system installati. Si tratta di istanze di specializzazioni della classe astratta **java.nio.file.spi.FileSystemProvider** (sempre introdotta con Java SE7). Questi sono molto importanti giacché la stragrande maggioranza dei metodi forniti dalla classe **Files** (descritta di seguito), sono delegati a questi provider. I provider installati sono caricati per mezzo del servizio di caricamento dei service provider fornito dalla classe **java.util.ServiceLoader**. I file system provider sono tipicamente installati inserendoli in un JAR posizionato nel classpath dell'applicazione o nella directory delle estensioni.

Una volta ottenuta una classe di tipo **FileSystem** (notare il singolare), più precisamente un'istanza di una sua specializzazione (**FileSystem** è una classe astratta), è possibile eseguire tutta una serie di metodi che restituiscono l'implementazione di una serie di oggetti molto utili.

Anche **FileSystem** è una sorta di factory in grado di restituire una serie di oggetti. In particolare:

- il metodo **getPath** si occupa di convertire path espressi per mezzo di stringhe e dipendenti dal sistema in oggetti di tipo **Path** utilizzabili per individuare e accedere a file e directory;
- il metodo **getPathMatcher** crea un oggetto di tipo **PathMatcher** (l'interfaccia **java.nio.file.PathMatcher** è stata introdotta con Java SE 7) in grado di eseguire operazioni di match su determinati path;
- il metodo **getFileStores** restituisce un oggetto iteratore sugli oggetti file-stores disponibili;
- il metodo **getUserPrincipalLookupService** restituisce gli **UserPrincipalLookupService** (la classe astratta **java.nio.file.attribute.UserPrincipalLookupService** è anch'essa stata introdotta con Java SE 7) che permettono di individuare utenti o gruppi attraverso i rispettivi nomi: questa funzionalità è necessaria per lavorare con sistemi operativi che permettono di specificare gli owner dei file;
- il metodo **newWatchService** permette di creare un oggetto **WatchService** (servizio di sorveglianza, **java.nio.file.WatchService**) che permette di osservare oggetti che possono cambiare.

Esempi

Di seguito è presentato un listato che mostra l'utilizzo di alcuni metodi base della classe astratta File System:

```
/**
 * The usual bad practice to test!
 * @param args
 */
public static void main(String[] args) {

  FileSystem fileSystem = FileSystems.getDefault();
  LOGGER.info(
     "Default file System:"+fileSystem.toString()+
     "- Is it open?"+fileSystem.isOpen());

  Iterable<Path> rootDirectories =
        fileSystem.getRootDirectories();
```

```
  for(Path currPath : rootDirectories) {
    LOGGER.info(
       "Root directory: '"+currPath.toString()+"'");
  }

  UserPrincipalLookupService userPrincipal =
     fileSystem.getUserPrincipalLookupService();

  LOGGER.info("User Principal:"+userPrincipal.toString());

  // N.B. the Default File System cannot be closed.
}
```

Listato 1. *Esempio di utilizzo delle classi FileSystems e FileSystem*

L'output generato su una macchina di test Windows 7 è:

```
Default file System: sun.nio.fs.WindowsFileSystem@169ca65- Is it
open? true
Root directory: 'C:\'
Root directory: 'D:\'
User Principal: sun.nio.fs.WindowsFileSystem$LookupService$1@196
8e23
```

Ed ecco un semplice esempio di utilizzo di **PathMatcher**:

```
/**
 * The usual bad practice to test!
 * @param args
 */
public static void main(String[] args) {

  if ((args == null) || (args.length == 0)) {
    LOGGER.warn("Please set the file path");
    return;
  }
```

```
    final String filePath = args[0];

    Path path =
       FileSystems.getDefault().getPath(filePath);

    LOGGER.info("path obj: "+path.toString());
    LOGGER.info("root:"+path.getRoot()+
           ", file name:"+path.getFileName());

    LOGGER.info(
      FileSystems.getDefault()
        .getPathMatcher("glob:*.java")
        .matches(path.getFileName()) );
}
```

Listato 2. *Esempio di utilizzo della funzione path matcher.*

```
Specificando il path: C:/prj/java_se_7/src/main/java/com/lvt/
javalessons/javase7/PathMatcherExample.java
```

L'output prodotto è:

```
path obj: C:\prj\java_se_7\src\main\java\com\lvt\javalessons\
javase7\PathMatcherExample.java
root:C:\, file name:PathMatcherExample.java
true
```

glob e regex

Il metodo **getPathMatcher** richiede un parametro che specifica la coppia (sintassi, pattern) separati dal carattere due punti, ossia

```
<sintassi>:<pattern>
```

Allo stato attuale sono previste due sintassi: **glob** e **regex**. La seconda, come è lecito attendersi, rappresenta le espressioni regolari (**REG**ular **EX**pression, introdotte con J2SE 1.4) e quindi la componente pattern è un'espressione regolare come definito dalla classe **java.util.regex.Pattern**. La sintassi **glob** invece è una

new entry data da una versione limitata e semplificata delle espressioni regolari. Qui di seguito sono riportati alcuni esempi:

`*.java`
indica un percorso che rappresenta un nome di file con estensione **.java**

`*.*`
indica nomi di file che contengono un carattere punto

`*.{java,class}`
indica nomi di file la cui estensione è **.java** o **.class**

`log.?`
indica nomi di file che cominciano con un nome "log." e che hanno un solo carattere di estensione

`/home/*/*`
indica un percorso **/home/<qualsiasi directory>/<qualsiasi directory>** tipicamente su piattaforma UNIX

`/home/**`
indica un percorso **/home/<qualsiasi combinazione di directory>** tipicamente su piattaforma UNIX

`C:\\*`
indica **C:\<qualsiasi directory>** su piattaforme Windows (l'equivalente stringa Java è "**C:***")

Per ulteriori dettagli circa la sintassi glob cfr. [Java Tut. F.O., 2012].

WatchService

Vediamo adesso un semplice esempio di utilizzo del servizio **WatchService**:

```
public class SimpleWatchService {
  // --------------- CONSTANTS SECTION ---------------

  /** logger */
  private static final Logger LOGGER   =
```

```
        Logger.getLogger(SimpleWatchService.class);

    // --------------- ATTRIBUTES SECTION ---------------

    // ----------------- METHODS SECTION -----------------

    /**
     * watch the given path
     * @param dirPath    directory to watch
     * @throws IOException a problem occurred
     * @throws InterruptedException
     */
    public static void   launchWatchService(String dirPath)
           throws  IOException, InterruptedException {

      FileSystem defaultFileSystem =
            FileSystems.getDefault();
      WatchService watchService =
            defaultFileSystem.newWatchService();

      Path path = defaultFileSystem.getPath(dirPath);
      // register the events to be notified
      path.register(
         watchService,
         ENTRY_CREATE, ENTRY_DELETE);

      boolean fine = false;
      int indx = 0;

      while (!fine) {
        LOGGER.info("Iteration:"+indx);
        // take the requested events. This is a blocking operation
        WatchKey watchKey = watchService.take();
        List<WatchEvent<?>> events = watchKey.pollEvents();
        LOGGER.info("--> Num Event:"+events.size());

        // goes through the list of events
        for(WatchEvent<?> currentEvent : events) {
```

```
        LOGGER.info("--> Event:"+currentEvent.kind()+
               " on \""+currentEvent.context()+"\"");

        if (currentEvent.kind().equals(ENTRY_CREATE)) {
          LOGGER.info(": do nothing");

        } else if (currentEvent.kind().equals(ENTRY_DELETE)) {
          LOGGER.info(": shutdown");
          fine = true;
        }
      }
      watchKey.reset();  // reset after processing

      indx++;
    }
  }

  /**
   * main
   * @param args command line parameters
   */
  public static void main(String[] args) {
    LOGGER.info("Launching the service...");
    try {
      launchWatchService("C:/tmp");

    } catch (IOException | InterruptedException e) {
      LOGGER.warn("Wooops... We have a problem " + e.getMessage());
    }
  }
}
```

Listato 3. *Esempio di WatchService.*

Funzionamento

Prima di procedere alla descrizione del funzionamento, riportiamo il diagramma delle classi della rete di interfacce che definiscono il servizio WatchService. La relativa visione dovrebbe chiarire alcuni aspetti di dettaglio.

La struttura di questa classe di esempio è abbastanza semplice. Il metodo launchWatchService ottiene il riferimento all'oggetto FileSystem di default con il quale genera una nuova istanza di tipo WatchService. Dopo aver creato un oggetto Path impostato alla directory da monitorare fornita come parametro al metodo, vi registra il WatchService (l'interfaccia Path estende Watchable che definisce questo metodo) specificando gli eventi che si intende ricevere (**path.register(watchService, ENTRY_CREATE, ENTRY_DELETE**), questi eventi sono definiti dalla classe java.io.file.StandardWatchEventKinds): la creazione di un nuovo entry e la cancellazione di un entry esistente. A questo punto l'esecuzione esegue un loop che contiene l'operazione bloccante di "presa" eventi (**WatchKey watchKey = watchService.take()**) seguita dall'operazione di lettura dei vari eventi (**watchKey.pollEvents()**). Se l'evento ricevuto è CREATE, allora si limita ad effettura l'output dei dati dell'evento, mentre se è un DELETE, esegue l'output e termina il loop e l'esecuzione.

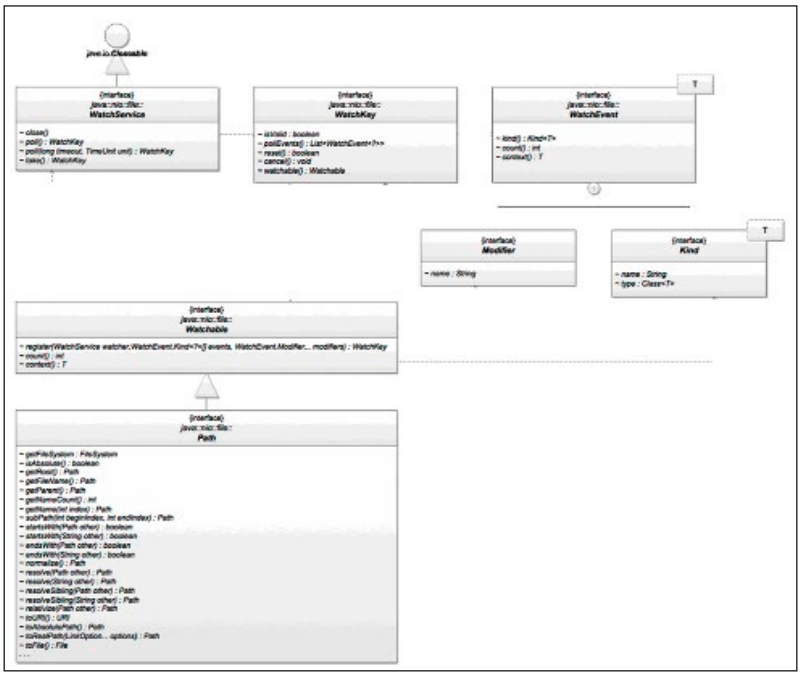


Figura 2. *Interfacce WatchService.*

Dopo aver lanciato il programma su piattaforma Windows 7 ed aver aperto Windows Explorer sulla directory **C:/tmp**, abbiamo creato una nuova directory, attraverso l'opzione **new folder** presente nel menu contestuale del tasto destro e quindi l'abbiamo ribattezzata in **Test**. Ecco l'output prodotto dal programma:

```
Launching the service...
Iteration:0
--> Num Event:1
--> Event:ENTRY_CREATE on "New folder" : do nothing
Iteration:1
--> Num Event:2
--> Event:ENTRY_DELETE on "New folder" : shutdown
--> Event:ENTRY_CREATE on "Test" : do nothing
```

A prima analisi l'output potrebbe sembrare un po' oscuro, ma da una seconda lettura è possibile evidenziare il seguente comportamento: il sistema operativo crea una sottodirectory con il nome di default (New folder). A seguito dell'operazione di cambiamento del nome, il sistema operativo dapprima rimuove la nuova directory "New Folder" e poi ne crea una nuova con il nome impostato: Test. Tuttavia, poiché è avvenuta una cancellazione, la procedure forza lo shutdown.

Le motivazioni alla base della classe Files

Java dispone di una classe per lavorare con i file sin dalla prima versione: **java.io.File**. Java SE 7 rilascia una nuova versione completamente rivista. Questo essenzialmente per i motivi che vediamo di seguito.

Prima di tutto, diversi metodi non generavano **eccezioni** significative in caso di fallimento. Ciò rendeva impossibile ottenere messaggi di errore utili. L'esempio più eclatante è la cancellazione di un file il cui fallimento risulta in un generico messaggio "delete fail", senza fornire ulteriori spiegazioni sulla natura del problema: file non esistente, insufficienti autorizzazioni, etc.

Un altro problema era dovuto al fatto che il metodo di **ridenominazione** non presentava un funzionamento coerente sulle varie piattaforme.

Poi, il supporto dei **link simbolici** era carente, così come il supporto per i **metadati**, mancavano all'appello i **permessi** dei file, del proprietario del file, degli attributi di sicurezza; l'accesso ai metadati del file era inefficiente.

Molti metodi forniti dalla classe **File** presentavano problemi di scalabilità. Richieste di elenchi di directory di grandi dimensioni ad un sever poteva causarne

un blocco. La richiesta della lista dei file di directory di grandi dimensioni spesso causava problemi di risorse di memoria, con conseguente negazione del servizio.

Non era possibile scrivere codice affidabile in grado di **traversare ricorsivamente alberi** di file e comportarsi adeguatamente in caso di presenza di link simbolici circolari (p.e. il link A si riferisce al link B che si riferisce al link C che a sua volta si riferisce al link A).

Link simbolici e interfaccia Path

La classe **java.nio.file.Files** presenta esclusivamente metodi statici: è a tutti gli effetti una utility class. Prima di proseguire nella descrizione, è importante illustrare due concetti abbastanza nuovi nelle libreria Java: **link simbolici** e interfaccia **Path**.

Tradizionalmente, tutti gli oggetti presenti nel file system erano o directory of file. Tuttavia, **molti sistemi operativi supportano la nozione di link simbolici** (**symbolic links**, detti anche **symlink** o **soft link**). Si tratta di file speciali che in realtà sono dei riferimenti ad altri file, un po' come alcuni collegamenti che si creano sui desktop windows. La nuova libreria **NIO** è in grado di gestire i link simbolici o automaticamente (risolvendo i vari link fino a raggiungere il file/directory), o richiedendo al client della libreria di specificare il comportamento da seguire (p.e. NOFOLLOW). Da notare che molti sistemi operativi supportano anche il concetto di **hard link**, ossia di **soft link** con ulteriori restrizioni, come per esempio: la destinazione del link deve esistere, non è possibile creare hard link relativi a directory, "volumi", partizioni e file system condivisi, e così via.

L'interfaccia Path è una dei principali entry point del package **java.nio.file** ed è una rappresentazione programmatica di un percorso nel file system. Un oggetto **Path** contiene il nome del file/directory e della lista delle directory utilizzate per costruire il percorso, ed è usato per esaminare, individuare e manipolare file. Gli oggetti **Path** riflettono le piattaforme sottostanti. Nel sistema operativo Solaris, un percorso utilizza la sintassi Solaris (**/home/joe/foo**) e in Microsoft Windows, un percorso utilizza la sintassi di Windows (**C:\home\joe\foo**). Un percorso non è indipendente dal sistema. Il file o la directory corrispondente al percorso potrebbe non esistere. È possibile creare un'istanza di **Path** e manipolarla in vari modi: è possibile aggiungere o estrarne parti, confrontarla con un altro percorso, etc. Quando necessario, è possibile utilizzare i metodi della classe **File** per verificare l'esistenza del file corrispondente al percorso, creare il file, aprirlo, cancellarlo, modificarne i permessi, e così via. Esiste anche la classe **java.nio.file.Paths** che include solo due metodi statici che si occupano di convertire URI e sequenze di stringhe in corrispondenti oggetti di **Path**.

La classe Files

Come scritto in precedenza, la classe **Files** contiene esclusivamente metodi statici la cui implementazione, nella quasi totalità dei casi è delegata al provider. Di seguito sono riportate delle brevi descrizioni dei metodi più interessanti forniti dalla classe **Files**. Da notare che il termine "file", secondo buone tradizione della teoria del file system, è utilizzato per indicare sia veri e propri file, sia directory indistintamente, giacché al livello di file system sono tutti comunque dei file.

copy

```
copy(Path, OutputStream) : long
copy(InputStream, Path, CopyOption[]) : long
copy(Path, Path, CopyOption[]) : Path
```

Questa serie di metodi permette di eseguire la copia dei file da un path iniziale a uno finale. Oltre alle versioni attese, come per esempio l'overloading con **InputStream**, l'ultima è molto interessante giacché prevede come parametri formali il path di partenza, quello di destinazione e una serie di opzioni, quali: **REPLACE_EXISTING**, **COPY_ATTRIBUTES** e **NOFOLLOW_LINK** che permettono di specificare il comportamento da seguire nei vari casi. Da notare che se il file source è un link simbolico, questo non viene seguito e quindi ciò che viene copiato è il link al file e non il file stesso.

Vediamo subito un esempio con un semplice programma che mostra la copia di file da un URI a una directory locale:

```
public class SimpleUrlCopy {

  // --------------- CONSTANTS SECTION ---------------
  /** logger */
  private static final Logger LOGGER   =
       Logger.getLogger(SimpleUrlCopy.class);

  // --------------- ATTRIBUTES SECTION ---------------

  // ----------------- METHODS SECTION -----------------

  /**
   * main
```

```
    * @param args command line parameters
    */
   public static void main(String[] args) {

     Path path = null;

     try {
       path = Files.createTempDirectory("mypdf");
     } catch (IOException ioe) {
       LOGGER.warn("We have a problem "+ioe);
     }

     if (path != null) {

       LOGGER.info("Create new temp directory:\""+path+"\"");
       URI u =
           URI.create(
             "http://www2.mokabyte.it/cms/"+
             "attachmentproviderservlet?attachId="+
             "16S-UNO-VKU-51T_7f000001_29536129_9cd098d4"+
             "_JavaVersionsHistory.pdf");

       try (InputStream in = u.toURL().openStream()) {
         Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);

       } catch (IOException ioe) {
         LOGGER.warn("We have a problem "+ioe);
       }
     }
   }

}
```

Listato 4. *Esempio di utilizzo del metodo copy.*

createDirectory/createDirectories

```
createDirectories(Path, FileAttribute[]) : Path
createDirectory(Path, FileAttribute[]) : Path
```

Questi metodi permettono di creare, rispettivamente, una directory e una directory in modo ricorsivo, ossia creando anche eventuali directory genitore non ancora presenti. Questi richiedono di specificare gli attributi del file (inteso come oggetto directory). **FileAttribute** è una nuova interfaccia che permette di specificare questi attributi come coppie (**nome, valore**). Qualora si tenti di specificare attributi non contemplati nella lista di quelli che è possibile impostare automaticamente durante la creazione della directory, si ottiene un'apposita eccezione (**UnsupportedOperationException**).

createFile

```
createFile(Path, FileAttribute[]) : Path
```

Tenta di generare un nuovo file vuoto, e termina con insuccesso nel caso in cui il file già esista. Come nel caso di **createDirectory**, anche in questo caso è possibile specificare i gli attributi file, con le medesime modalità e avvertenze.

createLink

```
createLink(Path, Path) : Path
createSymbolicLink(Path, Path, FileAttribute[]) : Path
```

Tentano di creare un **link** a un file esistente. Il primo metodo cerca di creare "hard link".

createTempDirectory/createTempFile

```
createTempDirectory(String, FileAttribute[]) : Path
createTempDirectory(Path, String, FileAttribute[]) : Path

createTempFile(String, String, FileAttribute[]) : Path
createTempFile(Path, String, String, FileAttribute[]) : Path
```

Questi metodi creano, rispettivamente, una directory **temporanea** o un file temporaneo combinando il path della directory e una stringa da usare come prefisso.

Delete

```
delete(Path) : void
```

Cerca di **eliminare** il file o la directory specificata. Se il path specificato è un link simbolico, allora viene eliminato il link e non il file stesso. Se invece si tratta

di una directory, allora, come di consueto, l'implementazione si assicura che questa sia vuota per poter procedere all'eliminazione; in caso contrario genera un'apposita eccezione (**DirectoryNotEmptyException**).

deleteIfExists

```
deleteIfExists(Path) : boolean
```

Elimina un file o directory **se esiste**. La differenza con il metodo precedente è che questo ritorna un valore booleano per specificare se l'operazione ha avuto successo o meno, mentre il metodo precedente non restituisce alcune valore in caso di successo, mentre in caso di errore (il file specificato non esiste), genera un'eccezione (**NoSuchFileException**).

exists

```
exists(Path, LinkOption[]) : boolean
```

Verifica se il file (o directory) specificato **esiste o meno**. In questo caso è possibile specificare delle opzioni di link per istruire il metodo su come comportarsi in caso di link. Per default, il metodo segue i vari link fino a raggiungere l'elemento puntato. Tuttavia, è possibile specificare opzioni quali **NOFOLLOW_LINKS** con ovvio significato.

getAttribute/getAttributeView

```
getAttribute(Path, String, LinkOption[]) : Object
getFileAttributeView(Path, Class, LinkOption[]) : V
```

Restituisce il valore dell'attributo o degli attributi specificati. Il comportamento in caso di link simbolici è esattamente lo stesso specificato per il metodo **exists**. Nel caso di **getAttributes**, questi (in maniera decisamente inattesa) vanno specificati per mezzo di una stringa suddivisi per mezzo di una virgola.

getFileStore

```
getFileStore(Path) : FileStore
```

Restituisce un oggetto di tipo **FileStore** che rappresenta il luogo (**store**, in italiano: "magazzino", "luogo di conservazione") dove il file è ubicato. Da notare che la classe astratta **java.nio.file.FileStore** è anch'essa una nuova feature introdotta con Java SE 7. Istanze delle specializzazioni di questa classe rappresentano

uno storage pool, un dispositivo, una partizione, un volume, un file system o altre implementazioni che costituiscano un file store.

getLastModifiedTime, getOwner, getPosixFilePermissions

```
getLastModifiedTime(Path, LinkOption[]) : FileTime
getOwner(Path, LinkOption[]) : UserPrincipal
getPosixFilePermissions(Path, LinkOption[]) :
Set<PosixFilePermission>
```

Questi metodi restituiscono rispettivamente:

- il **timestamp** dell'ultima modifica avvenuta sul file/directory specificati attraverso un'istanza della una nuova classe final: **java.nio.file.attribute.FileTime**;
- il possessore del file (specificato per mezzo di implementazioni della nuova interfaccia **java.nio.file.attribute.UserPrincipal**, che specializza l'interfaccia **java.nio.file.attribute**;
- l'insieme dei permessi **POSIX** (Portable Operating System Interface, interfaccia portatile del sistema operativo) del file. Si tratta di una famiglia di standard di permessi specificati e gestiti dalla IEEE (IEEE Std 1003.1-1988, rilasciata nel 1988) al fine di mantenere una certa compatibilità tra i vari sistemi operativi. Ogni elemento di questo insieme è specificato per mezzo del nuovo tipo enumerato: **java.nio.file.attribute.PosixFilePermission** i cui elementi sono: **GROUP_EXECUTE**, **GROUP_READ**, **GROUP_WRITE**, **OTHERS_EXECUTE**, **OTHERS_READ**, **OTHERS_WRITE**, **OWNER_EXECUTE**, **OWNER_READ**, **OWNER_WRITE**.

setLastModifiedTime, setOwner, setPosixFilePermissions

```
setLastModifiedTime(Path, FileTime) : Path
setOwner(Path, UserPrincipal) : Path
setPosixFilePermissions(Path, Set) : Path
```

Si tratta dei **metodi di impostazione** dei valori corrispondenti ai metodi **get** succitati.

isDirectory, isExecutable, isHidden, isReadable, isRegularFile, isSymbolicLink, isWritable

```
isDirectory(Path, LinkOption[]) : boolean
isExecutable(Path) : boolean
```

```
isHidden(Path) : boolean
isReadable(Path) : boolean
isRegularFile(Path, LinkOption[]) : boolean
isSymbolicLink(Path) : boolean
isWritable(Path) : boolean
```

Questi metodi permettono di capire se un determinato file (o directory) è del tipo (isDirectory, isSymbolicLink) **o possiede** la **proprietà** (isExecutable, isHidden, etc.) specificata dal metodo.

isSameFile

```
isSameFile(Path, Path) : boolean
```

Permette di **comparare** i file/directory specificati dai rispettivi path. Restituisce un valore **true** se si tratta degli stessi oggetti.

move

```
move(Path, Path, CopyOption[]) : Path
```

Permette di **spostare o rinominare** il file (o directory) specificato nel path di destinazione. Questo metodo fallisce nel caso in cui esista già un file target differente da quello che si intendeva "muovere". Qualora il file sia un link simbolico, viene mosso il link e non il file puntato. In questo caso è possibile specificare le seguenti opzioni **REPLACE_EXISTING** e **ATOMIC_MOVE** con la semantica attesa.

probeContentType

```
probeContentType(Path) : String
```

Sonda il contenuto del file per identificarne il tipo, che viene restituito secondo lo standard **MIME** (Multipurpose Internet Mail Extension). Questo metodo utilizza le implementazioni della classe astratta **java.nio.file.spi.FileTypeDetector** (sempre introdotta con Java SE7) fornita dal service provider per sondare il file dato per determinare il tipo. Chiaramente, i vari provider possono utilizzare diverse strategie, come leggere le estensioni dei file, individuare determinati pattern nel file, etc. Il funzionamento prevede l'invocazione di ciascun rivelatore di tipo di file che si occupa di sondare se il file è del "proprio" tipo o meno (una implementazione del pattern **Chain of Responsibility**). Nel caso in cui il test abbia

esito affermativo, allora restituisce il tipo (stringa MIME), altrimenti prosegue. Se alla fine di questo ciclo, il file non è stato riconosciuto da nessuno dei rivelatori installati, allora viene invocato il detector di default per cercare di "indovinare" il tipo di contenuto.

readAllBytes, readAllLines

```
readAllBytes(Path) : byte[]
readAllLines(Path, Charset) : List<String>
```

Questi metodi si occupano di **caricare in memoria** l'intero file memorizzandolo, rispettivamente, in un **array di byte** e in una **lista di stringhe**. Da notare che questi metodi possono lanciare un **OutOfMemoryError** qualora si ecceda lo spazio a disposizione (heap).

Size

```
size(Path) : long
```

Restituisce le **dimensioni** del file specificato in termini di byte.

walkFileTree

```
walkFileTree(Path, FileVisitor) : Path
walkFileTree(Path, Set, int, FileVisitor) : Path
```

Si tratta di una serie di metodi molto potenti che permette di effettuare la **navigazione di una parte** (albero) del file system a partire dal nodo (file) dato. L'albero è attraversato secondo l'algoritmo **depth-first** (profondità per prima, continua a visitare i nodi prima di tornare sui propri passi e tentare le altre direzioni). Importante parametro di questo metodo è un'implementazione dell'interfaccia **java.nio.file.FileVisitor<T>** che permette di ricevere notifiche ogni volta che viene individuato un nuovo elemento dell'albero (per informazioni di dettaglio relative al pattern del visitatore si rimanda all'apposita appendice).

Le notifiche sono relative ai seguenti eventi:

- in procinto di visitare una directory (**preVisitDirectory**);
- appena terminata la visita di una directory (**postVisitDirectory**);
- individuazione di un nuovo file (**visitFile**);
- fallimento di visita di un file (**visitFileFailed**).

I metodi invocati della classe visitor sono tipicamente dotati di una serie di attributi base (**BasicFileAttribute**) e permettono di stabilire se proseguire o meno

la visita dell'albero. In particolare, possono restituire uno dei seguenti elementi dell'enumeration **java.nio.file.FileVisitResult**: **CONTINUE**, **SKIP_SIBLINGS**, **SKIP_SUBTREE**, **TERMINATE** i cui significati sono ovvi.

Da notare che, se durante la traversata della struttura qualcosa va storto e quindi vi è una **IOException**, viene invocato il metodo **visitFileFailed** del visitor, il quale deve restituire un elemento del **FileVisitResult** e quindi stabilire se proseguire o terminare.

```
public class WalkFileTreeExample {

  // --------------- CONSTANTS SECTION ---------------

  /** logger */
  private static final Logger LOGGER   =
        Logger.getLogger(WalkFileTreeExample.class);

  // --------------- ATTRIBUTES SECTION ---------------

  // ----------------- METHODS SECTION -----------------

  /**
   * Walk the given file
   * @filePath path of the file tree to walk
   */
  public static void walkGivenFileThree(String filePath) {

    if ((filePath == null) || (filePath.isEmpty()) ) {
      return;
    }

    LOGGER.info(
       ">>>>>Starting the walk of the file tree:"+
       filePath);

    Path start = Paths.get(filePath);
    try {
      Files.walkFileTree(
         start,
```

```
        new SimpleFileVisitor<Path>() );

    } catch (IOException ioe) {
      LOGGER.warn("We have a problem:"+ioe);
    }

    LOGGER.info(
       ">>>>>Terminating the walk of the file tree:"+
       filePath);
  }

  // -------------- INNER CLASS ------------------
  // Implements the File Visitor interface to receive
  // notification during the visit
  // -------------------------------------------------

  private static class      SimpleFileVisitor<T>
          implements FileVisitor<T> {

    @Override
    public FileVisitResult postVisitDirectory(
               T dir,
               IOException exc)
        throws          IOException {


      LOGGER.info("Pre-Visiting:"+dir.toString());

      return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(
               T file,
               BasicFileAttributes attrs)
        throws IOException {
      return FileVisitResult.CONTINUE;
    }
```

```
    @Override
    public FileVisitResult preVisitDirectory(
                T dir,
                BasicFileAttributes attrs)
            throws          IOException {

      return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(
                T file,
                IOException exc)
            throws          IOException {

      return FileVisitResult.CONTINUE;
    }
  }

  /**
   * main
   * @param args Command line parameters
   */
  public static void main(String[] args) {
    walkGivenFileThree("C:/java_dev/jdk1.7.0_02");

  }
} // --- end of class ---
```

Listato 5. *Esempio di utilizzo del metodo WalkFileTree.*

write

```
write(Path, byte[], OpenOption[]) : Path
write(Path, Iterable, Charset, OpenOption[]) : Path
```

Permette di **scrivere** dei byte o delle linee di testo (a seconda dello overloading del metodo utilizzato) nel file specificato. In questo caso è possibile specificare dei

parametri di scrittura del file, come per esempio **APPEND**, **CREATE_NEW**, e così via.

Oltre a questi metodi, ve ne è un'intera una serie per ottenere oggetti **BufferedReader/Writer**, **Channel**, **Stream** per poter manipolare i file.

E anche qui, la cosa migliore è vedere un esempio di utilizzo di alcuni metodi dell'interfaccia **Files**:

```
public class AttributesSample {

  // --------------- CONSTANTS SECTION ---------------

  /** logger */
  private static final Logger LOGGER   =
        Logger.getLogger(AttributesSample.class);

  // --------------- ATTRIBUTES SECTION ---------------

  // ----------------- METHODS SECTION -----------------

  public static void main(String[] args) {

    final String THIS_SOURCE_FILE =
       "C:/prj/java_se_7/src/resource/excel_worksheet.xlsx";

    Path path = FileSystems.getDefault().getPath(THIS_SOURCE_
FILE);

    try {
      LOGGER.info(
         "Last modified time:"+
         Files.getLastModifiedTime(path));
      LOGGER.info(
         "Owner:"+
         Files.getOwner(path));
      LOGGER.info(
         "Size:"+
         Files.size(path));
      LOGGER.info(
```

```
            "Type:"+
            Files.probeContentType(path));

        } catch (IOException ioe) {
          LOGGER.warn("We have a problem:"+ioe.getMessage());
        }

      }
    } // --- end of class ---
```

Listato 6. *Esempio di utilizzo dei metodi per reperimento degli attributi di un file.*

Di seguito è riportato l'output generato dalla classe precedente, epurato dalle informazioni specifiche del log:

```
Last modified time:2012-07-06T22:01:26.315429Z
Owner:Luca-Laptop-1\Luca (User)
Size:8746
Type:application/vnd.openxmlformats-officedocument.spreadsheetml.
sheet
```

DirectoryStream e SecureDirectoryStream

Java SE 7 fornisce le due interfacce **java.nio.file.DirectoryStream** e la sua specializzazione **java.nio.file.SecureDirectoryStream**.

DirectoryStream

DirectoryStream definisce il comportamento di classi che si occupano di scorrere le voci in una directory utilizzabile in un costrutto **for-each**. Un oggetto di tipo **DirectoryStream** viene aperto al momento della creazione ed è chiuso invocando il metodo **close**. Ciò si può fare anche automaticamente per mezzo del nuovo costrutto **try-with-resource**. L'interfaccia **DirectoryStream**, estende le interfacce **AutoCloseable** e **Closeable** (oltre ovviamente **Iterable**). La chiusura di un oggetto **DirectoryStream** genera il rilascio di tutte le risorse associate allo stream. La mancata chiusura, come al solito, può causare i classici leak.

Le motivazioni alla base dell'implementazione di questa interfaccia e delle corrispondenti implementazioni è di consentire di implementare oggetti in grado di presentare un buon livello di scalabilità lavorando su directory di grandi dimensioni, dando luogo ad un utilizzo parsimonioso delle risorse, con tempi di

risposta accettabili. Inoltre, questa interfaccia include la possibilità di eseguire il filtro delle voci della directory. Istanze di questa interfaccia si ottengono invocando il metodo **Files.newDirectoryStream**.

Vediamo di seguito un semplice esempio di utilizzo di **DirectoryStream**:

```
public class SimpleDirectoryStream {

  // --------------- CONSTANTS SECTION ---------------

  /** logger */
  private static final Logger LOGGER = Logger
      .getLogger(SimpleDirectoryStream.class);

  // --------------- ATTRIBUTES SECTION ---------------

  // ----------------- METHODS SECTION -----------------

  /**
   * Create a list with the files located in the given directory
   * @param dir   directory where to look
   * @return  list with the files located in the given directory
   * @throws IOException  a serious problem happened
   */
  public static List<Path> listSourceFiles(String dir)
           throws     IOException {

    if ((dir == null) || (dir.isEmpty())) {
      return null;
    }

    Path dirPath = FileSystems.getDefault().getPath(dir);
    List<Path> result = new ArrayList<>();

    // build a list with all files
    try (DirectoryStream<Path> stream =
        Files.newDirectoryStream(dirPath,"*.{c,java}")) {
      for (Path entry : stream) {
        result.add(entry);
```

```
      }
    }

    return result;
  }

  /**
   * main
   * @param args
   */
  public static void main(String[] args) {

    List<Path> paths;
    try {
      paths = listSourceFiles(
          "C:/prj/java_se_7/src/main/java/"+
          "com/lvt/javalessons/javase7/nio");
      for (Path currPath : paths) {
        LOGGER.info(currPath);
      }
    } catch (IOException ioe) {
      LOGGER.warn("We have a problem " +
            ioe.getMessage());
    }

  }
}
```

Listato 7. *Esempio di utillizzo di DirectoryStream*

SecureDirectoryStream

Oggetti di tipo **SecureDirectoryStream** sono pensati per un utilizzo da applicazioni sofisticate o sensibili dal punto di vista della **sicurezza** che necessitano di operare su alberi di file o operare su directory in assenza di **race-condition** (condizioni di competizione). In questo contesto, race condition insorgono per via del fatto che, tipicamente, sequenze di operazioni su file non avvengo atomicamente. Tutte le operazioni sui file definite da questa interfaccia richiedono di specificare un percorso relativo alla directory aperta.

L'implementazione di **SecureDirectoryStream** richiede il supporto da parte del sistema operativo sottostante. Qualora un'implementazione è in grado di supportare questa funzionalità il **DirectoryStream** restituito dal metodo **newDirectoryStream** è un **SecureDirectoryStream** che quindi richiede un **cast** esplicito per poter richiamare i metodi definiti da questa interfaccia.

SCTP

Un altro aggiornamento della libreria NIO è dato dal supporto dello **Stream Control Transmission Procotol** (**SCTP**, Protocollo di trasmissione su stream controllato, cfr. [SCTP, 2004]). Si tratta di un protocollo di trasporto che rappresenta un ibrido tra i popolari protocolli **TCP** (Transmission Control Protocol, protocollo a controllo di trasmissione) e **UDP** (User Datagram Protocol, protocollo di datagrammi utente). In particolare è orientato al messaggio come UDP (e non alla sessione) e, allo stesso tempo, assicura un trasporto affidabile e nella corretta sequenza con controllo di congestione come il TCP e Sockets Direct Protocol. Per capirne la relativa importanza basti considerare soluzioni quali le cache distribuite: ci sono una serie di nodi che partecipano al cluster che devono essere mantenuti in synch tra loro. Pertanto, il protocollo selezionato riveste un'importanza cruciale: da un lato è necessario disporre di un protocollo affidabile (quindi TCP), dall'altro è fondamentale poter contare su un protocollo leggero e su una latenza ridotta al minimo (quindi UDP). Come si risolveva questo problema in passato? Semplice, si selezionava il protocollo UDP e quindi si costruivano dei servizi superiori per cercarlo di renderlo più affidabile. Tutto ciò non dovrebbe essere più necessario giacché SCTP è proprio questo: il compromesso tra TCP e UDP. Le caratteristiche fondamentali di questo protocollo sono quelle descritte di seguito.

Message Framing

Significa "definizione/circoscrizione di messaggio" durante la trasmissione e ciò implica che la dimensione dei messaggi è preservata nell'invio nel canale. Pertanto se un client invia due messaggi a un sever, uno di dimensione X ed uno di dimensione Y, questi messaggi (entro certi limiti) resteranno identici nel canale, e quindi il server eseguirà due letture di dimensioni X e Y rispettivamente. Ciò in effetti è quello che avviene con il protocollo UDP. Mentre in TCP, orientato alla sessione, funziona su streaming di byte.

Servizio di trasporto affidabile

Questa è una delle caratteristiche molto interessanti del protocollo, in quanto, sebbene la strategia di funzionamento sia più simile a UDP, STPC esegue una

trasmissione affidabile proprio come TCP. Questo è ottenuto tramite la definizione di opportuni servizi che si occupano di garantire l'affidabilità alla comunicazione orientata al messaggio.

Mantenimento dell'ordine

Possibilità di richiedere la consegna dei messaggi nella stessa sequenza di emissione.

Multi-homing

Questa caratteristica fornisce alle applicazioni una maggiore disponibilità di quelle che utilizzano il protocollo TCP. Un server funziona **multi-homing** quando è dotato di più interfacce di rete e quindi più indirizzi IP che possono essere utilizzati per indirizzarlo. In TCP, una connessione si riferisce a un canale tra due punti (socket tra le interfacce dei due host). SCTP introduce il concetto di **associazione** che esiste tra due host (associazione tra due end-point), ma che è in grado di **collaborare** con **interfacce multiple** ad ogni host (ogni endpoint può essere rappresentato da più indirizzi). Ciò fa sì che **SCTP** possa lavorare in configurazioni ridondanti e quindi sia in grado di **assicurare un failover** trasparente. In particolare, SCTP controlla i percorsi delle associazioni con un predefinito servizio di heartbeat (controllo del battito cardiaco), ed è quindi in grado di dirottare il traffico su un percorso alternativo qualora rilevi un errore di percorso. Tutto ciò è ovviamente trasparente per le applicazioni.

Multi-Streaming

Le associazioni SCTP sono simili alle connessioni TCP con la differenza che SCTP è in grado di supportare diversi stream (flussi) all'interno di una medesima associazione. Quindi è possibile **partizionare** i dati **in diversi stream** indipendenti. Questo è ottenuto facendo sì che a ciascun stream sia assegnato un numero di identificazione univoco riportato all'interno dei corrispondenti pacchetti trasmessi. Multi-streaming è importante perché fa sì che un flusso bloccato (per esempio, uno stream in attesa di ritrasmissione per via della perdita di un pacchetto) non influisca su gli altri flussi presenti sulla medesima associazione. Questo problema, presente in TCP, è tipicamente indicato con il nome di **head-of-line blocking** (blocco di inizio linea) che quindi non si manifesta in SCTP. Si consideri un multi-streaming server HTTP. Questo potrebbe portare a una migliore interattività giacche' richieste multiple potrebbero venir risolte attraverso diversi stream indipendenti di una medesima associazione. Questa funzionalità permetterebbe di parallelizzare le risposte, e anche qualora non dovesse portare a migliori

performance, potrebbe sicuramente migliorare l'esperienza utente: il browser potrebbe contemporaneamente caricare il codice HTML e le immagini grafiche.

Come è lecito attendersi, la API Java si basa sull'architettura NIO: in questo modo le applicazioni che necessitano di utilizzare il protocollo possono sfruttare servizio di I/O non bloccanti in multi-plexing. Le nuove classi ed interfacce atte a implementare il supporto del protocollo SCTP secondo l'architettura **NIO** sono state incluse nel package **com.sun.nio.sctp** invece che in un package del tipo **java.nio.channels.sctp**. Ciò perché, sebbene l'API e l'implementazione siano pienamente supportate e pubblicamente accessibili, ancora non fanno ancora parte della piattaforma Java SE. In altre parole, sono in attesa di raggiungere un grado di maturità superiore.

La parte fondamentale del nuovo package consiste nei tre nuovi tipi di canale (**SctpChannel**, **SctpServerChannel**, **SctpMultiChannel**) che possono essere suddivisi nei seguenti due gruppi logici.

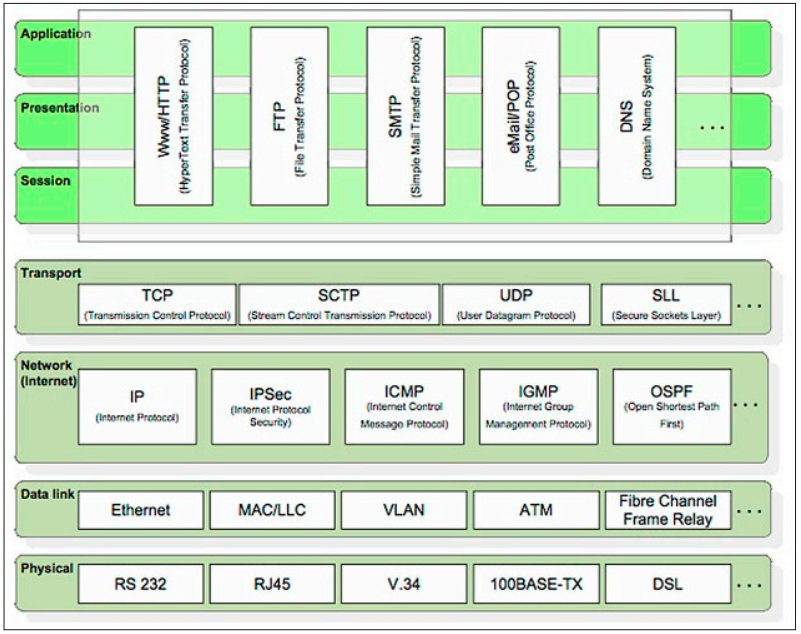


Figura 3. *Rappresentazione semplificata del classico modello OSI (Open Systems Interconnection).*

Il primo presenta una semantica simile a TCP, **SctpChannel** e **SctpServerChannel**. Un **SctpChannel** può controllare solo una singola associazione: invio e ricezione di dati da e verso un singolo endpoint. **SctpServerChannel** ascolta e accetta nuove associazioni richieste al proprio socket.

Il secondo è composto di soli **SctpMultiChannel**. Le istanze di questo tipo di canale possono controllare più associazioni, pertanto, rendono possibile inviare e ricevere dati da e verso svariati endpoint diversi.

Conclusione

In questo articolo abbiamo esaminato gli aggiornamenti dell'API NIO introdotti con JavaSE7. In particolare, abbiamo presentato una serie di cambiamenti abbastanza importanti tanto che la nuova versione è stata ribattezzata NIO2. Questi sono condensati nei nuovi package: **java.nio.file**, **java.nio.file.attribute** e **java.nio.file.spi**. Gli obiettivi primari di questa release consistevano nell'aumentare l'indipendenza dalla piattaforma di esecuzione, fornire ulteriori servizi di alto livello e aggiungere il supporto per metadati e collegamenti simbolici**.** Le nuove classi (soprattutto **Files**) forniscono una serie di servizi molto utili per la gestione dei file, inclusi "macro" servizi come lo spostamento dei file, la copia, e così via. . Per quanto concerne il supporto per i metadati, è stato introdotto il package **java.nio.file.attribute** allo scopo di fornire un supporto avanzato ai metadata dei.

Java NIO2 inoltre fornisce un supporto esteso ai link simbolici. Anche se ciò era parzialmente possibile con le versioni precedenti, vi erano alcuni problemi: per esempio non era immediato, e spesso neanche completamente possibile, scrivere del codice robusto per la navigazione ricorsiva e la gestione dei link che generavano cicli di link simbolici.

La stragrande maggioranza di queste feature sono state ottenute per mezzo dell'integrazione di servizi nativi forniti dalla piattaforma di esecuzione.

Ora molte applicazioni esistenti prevedono del codice per così dire "legacy" per l'ottenimento di questi servizi implementati con le versioni precedenti di questa libreria. Questo codice, per forza di cose, nella maggioranza dei casi prevede livelli di scalabilità e performance sicuramente inferiori a quelli ottenibili dalle nuove classi che sfruttano pesantemente i servizi offerti dai sistemi operativi di produzione ed implementazioni macchina.

Una domanda interessante da porsi è se valga la pena migrare tale codice "legacy", magari non ottimale ma ben testato, sulla nuova libreria IO o restare ancorati alla classe **java.io.File**. Chiaramente non esiste una risposta univoca e la valutazione va eseguita caso per caso. Tuttavia può essere molto sensato mettere in

programma questa migrazione qualora l'utilizzo dei file sia una componente importante dell'applicazione e vi sia la necessità di migliorare uno o più dei seguenti fattori: robustezza, performance e scalabilità. La migrazione verso NIO2 dovrebbe, tra l'altro, eliminare importanti porzioni di codice.

Capitolo 5
Le feature restanti

Introduzione

In questo capitolo presentiamo le restanti feature introdotte con Java SE7. Si tratta di una serie di funzionalità che seppure molto importanti in determinati contesti, non possono di certo essere considerate caratterizzanti per questa nuova release.

Le feature presentate in questo capitolo sono raggruppate nei seguenti gruppi:

- **internazionalizzazione** (aggiornamento alla versione Unicode 6.0, supporto estensibile per i codice standard delle valute, sdoppiamento del Locale);
- **UI** (look and feel Nimbus, JLayer, finestre traslucenti, OpenType/CFF font, NumericShaper, XRender per Java 2D);
- nuova versione **JDBC** (4.1);
- supporto della **crittografia** basata sulle curve ellittiche (ECC, Elliptic Curve Cryptography).

Internazionalizzazione

Unicode 6.0.0

Con Java SE 7 è stato aggiornato il supporto allo Unicode versione **6.0** (cfr. [Unicode 6, 2011]). La storia tra Java e Unicode è basta sui seguenti rilasci:

- le versioni precedenti alla 1.1 supportavano Unicode 1.1.5;
- con JDK 1.1 si è passati a Unicode 2.0;
- JDK 1.1.7 ha aggiunto il supporto a Unicode 2.1;
- con Java SE 1.4 si è passati a Unicode 3.0;
- Java SE 5.0 supporta Unicode 4.0;
- Java SE 7 supporta Unicode 6.0 (release iniziali di Java SE 7 supportavano la versione 5.1.0).

La versione Unicode 6.0 è una vera e propria major release con numerosi cambiamenti. Tra le variazioni più interessanti, vale la pena menzionare l'aggiunta di **2088 caratteri**. Tra questi ricordiamo:

- **oltre 1000 simboli aggiuntivi** tra i quali primeggiano i simboli **emoji** (**emoji** è il termine Giapponese per indicare caratteri "immagine", tipicamente 12 x 12

pixel, ossia emoticons largamente utilizzati nei messaggi elettronici giapponesi e nelle pagine web), che sono particolarmente importanti per i telefoni cellulari;
- il nuovo **simbolo ufficiale** indiano per la valuta **Rupia indiana**;
- 222 ulteriori ideogrammi unificati CJK di uso comune in Cina, Taiwan e Giappone;
- **603 caratteri supplementari** per il supporto ad alcune **lingue africane**, comprese le estensioni al Tifinagh, alla scrittura etiopica, e al Bamum;
- tre script aggiuntivi (Mandaic, Batak, e Brahmi.

Inoltre Unicode 6.0 include l'**aggiunta di nuove proprietà e file di dati**:
- EmojiSources.txt utilizzato per associare i simboli emoji alla loro sorgente telco originale giapponese;
- aggiunta di due proprietà provvisorie per il supporto di script indiani (IndicMatraCategory e IndicSyllabicCategory);
- l'**estensione per i dati provvisori** di script per l'uso nella segmentazione, espressioni regolari, e rilevamento spoof.

Non mancano poi inevitabili **correzioni** (**bug fix**) delle proprietà dei caratteri per i caratteri esistenti:
- 36 caratteri non-CJK;
- numerosi miglioramenti alle proprietà provvisorie per ideogrammi CJK Unified;
- aggiornamenti di formato per molti tag sorgenti della normativa IRG, per sincronizzare meglio con ISO / IEC 10646 (vedi UAX # 38, Unicode Database Han, per i dettagli); e in generale si fornisce miglioramenti al formato, inclusi grafici per ideogrammi di compatibilità CJK.

Supporto estensibile per i codice standard delle valute

L'**ISO 4217** (cfr. [ISO 4217, 2008]) è uno standard della International Standards Organization (organizzazione internazionale degli standard), che delinea i designatori di valuta, i codici dei Paesi (alfa e numerico), e riferimenti alle unità minori nelle seguenti tre tabelle.

La **Tabella A.1** include i codici delle valute correnti e della lista dei fondi. Di seguito sono riportate due righe che è possibile trovare in tale tabella.

Entity	Currency	Alphabetic code	Numeric code	Minor Unit
ITALY	Euro	EUR	978	2
ZZ08_Gold	Gold	XAU	959	N.A.

La **Tabella A.2** è ralativa ai codici dei fondi registrati e gestiti dall'agenzia ed include valori come:

```
UNITED STATES, US Dollar, Same Day, USS, 998, 2;
```

Per finire, la **Tabella A.3** include l'elenco dei codici relativi alle denominazioni storiche delle valute e dei fondi.

Un elemento fondamentale di questa feature è che tale supporto è estensibile: i **codici standard** possono essere **sovrascritti**. A tal fine è necessario creare id file di proprietà **currency.properties** e memorizzarlo nella directory:

```
<JAVA_HOME>/lib/currency.properties.
```

Il file deve essere organizzato in forma di coppie **key,value** (chiave, valore) relativi allo standard ISO 3166 dei codici nazioni, e allo standard ISO 4217 delle valute. L'elemento value in realtà consiste in tre elementi dei valori ISO 4217 delle valute separati per mezzo di una virgola: codice alfabetico, codice numerico, e numero di unità minori. Per esempio, il key, value del Giappone potrebbe prevedere **JP=JPZ,999,0**

Come è lecito attendersi, anche la classe **java.util.Currency** è stata **aggiornata** di conseguenza attraverso l'aggiunta dei seguenti metodi:

`getAvailableCurrencies`
restituisce l'insieme delle valute disponibili.

`getNumericCode`
restituisce il codice numerico ISO 4217 della valuta.

`getDisplayName`
restituisce il nome idoneo per visualizzare la valuta in base al "locale" impostato.

`getDisplayName(Locale)`
restituisce il nome idoneo per visualizzare la valuta in base al "locale" specificato.

E vediamo di seguito l'utilizzo del metodo **getAvailableCurrencies**, quanto mai atteso da chi lavora in finanza:

```
public class CurrencyExample {

  // ---------------- CONSTANTS SECTION ----------------
  /** logger */
  private static final Logger LOGGER =
       Logger.getLogger(CurrencyExample.class);

  // ---------------- ATTRIBUTES SECTION ---------------

  // ---------------- METHODS SECTION ------------------

  /**
   * Poor way to test classes!
   *
   * @param args
   */
  public static void main(String[] args) {

    Set<Currency> currencies =
          Currency.getAvailableCurrencies();

    for (Currency ccy : currencies) {
    LOGGER.info(
         ccy.getCurrencyCode()+"\t"+
         ccy.getNumericCode()+"\t"+
         ccy.getDisplayName());
    }
  }
}
```

Listato 1. *Esempio di utilizzo delle nuove feature della classe Currency.*

L'esecuzione del precedente listato, al netto delle informazioni del log, produce un output come il frammento riportato di seguito.

```
UYU    858    Uruguayan       Peso
MKD    807    Macedonian      Denar
TRL    792    Turkish         Lira (1922-2005)
```

```
SYP    760    Syrian          Pound
ISK    352    Icelandic       Króna
GWP    624    Guinea-Bissau   Peso
NIO    558    Nicaraguan      Córdoba
LVL    428    Latvian         Lats
XPT    962    Platinum
AUD    36     Australian      Dollar
XAU    959    Gold
SEK    752    Swedish         Krona
LSL    426    Lesotho         Loti
```

Internazionalizzazione e localizzazione

Prima di proseguire oltre, è importante chiarire alcuni concetti di base, come **internazionalizzazione** e **localizzazione**.

Con il termine Internationalization (internazionalizzazione) ci si riferisce ad una specifica fase del processo di disegno ed implementazione di un'applicazione il cui obiettivo consiste nell'assicurarsi che questa possa essere adattata alle varie lingue e regioni, senza modifiche tecniche. Per essere precisi, agli inizi il problema da risolvere era quello di assicurarsi che una data applicazione presentasse presentare un'esperienza utente confortevole per le varie nazioni in cui se ne prevedeva l'utilizzo. Con il passare del tempo, la preoccupazione è divenuta sempre più quella di far in modo che utenti di diverse culture possano utilizzare un sistema secondo le proprie preferenze linguistiche, di formattazione, tipografiche, e così via. Spesso **il termine internazionalizzazione è abbreviato come i18n**, per il semplice fatto che ci sono 18 lettere tra la prima "i" e l'ultimo "n" di Internationalization.

La localizzazione è il processo di adattamento di un software per una specifica regione o lingua con l'aggiunta di componenti specifici, la variazione dei formati e la traduzione del testo. **Il termine localizzazione è spesso abbreviato in l10n**, perché ci sono 10 lettere tra la "l" e la "n" (Localization).

Il compito principale della localizzazione è più complesso di quello che si potrebbe pensare. Infatti, include la traduzione degli elementi dell'interfaccia utente, della documentazione prodotta, delle validazione di acquisizione di documenti, fogli elettronici, e così via. La localizzazione coinvolge non solo cambiare la lingua di interazione, ma anche altri cambiamenti importanti come la visualizzazione-acquisizione di numeri, date, valuta ecc. Come se non bastasse, in alcuni contesti, la localizzazione piò richiedere di variare altri tipi di dati, come ad esempio suoni e immagini.

Un programma propriamente internazionalizzato è quindi in grado di visualizzare/acquisire/produrre informazioni in modo diverso in tutto il mondo o meglio a seconda delle specifiche impostazioni dell'utente. Per esempio, lo stesso programma può mostrare messaggi in italiano per utenti italiani, in Inglese per i sudditi della regina, e così via.

In Java **java.util.Locale** è la classe che gioca un ruolo fondamentale per l'Internazionalizzazione e localizzazione del software. Un oggetto Locale è un identificatore per una particolare combinazione di lingua e regione. Se un oggetto è in grado di variare il proprio comportamento in base alle impostazioni internazionali, si dice che è locale-sensitive. Ad esempio, la classe NumberFormat è locale-sensibile, il formato del numero restituito dipende dal Locale specificato (cfr. esempio seguente). Così NumberFormat può restituire un numero come 902.300 (Italia), 902,300 (Germania) e 902.300 (Stati Uniti). Oggetti Locale sono identificatori unici. Il vero lavoro, come la formattazione, la variazione delle regole di acquisizione dell'input, è eseguita dagli oggetti in grado di operare con locale.

Sdoppiamento del "locale"

Con Java SE7 è possibile impostare due locale di default in modo indipendente per i due diversi utilizzi:

- **format setting** (impostazione di formato) utilizzato per formattare le risorse;
- **display setting** (impostazione di visualizzazione) utilizzato dai menu e dalle finestre di dialogo.

Il nuovo metodo **public static Locale getDefault(Locale.Category category)** della classe **java.util.Locale** prende come unico parametro **Locale.Category**: si tratta di un nuovo enumeratore che prevede i due literal **FORMAT** e **DISPLAY**. Passando il **FORMAT**, il metodo restituisce le impostazioni internazionali predefinite per le risorse di formattazione. Analogamente, fornendo il valore **DISPLAY** restituisce il locale predefinito utilizzato dall'interfaccia utente. Il metodo **setDefault(Locale.Category category, Locale newLocale)** è utilizzato per impostare la localizzazione per la categoria specificata.

```
public class LocaleSample {

  // ---------------- CONSTANTS SECTION ----------------
  /** logger */
  private static final Logger LOGGER =
```

```
    Logger.getLogger(LocaleSample.class);

// ---------------- ATTRIBUTES SECTION ---------------

// ----------------- METHODS SECTION -----------------


/**
 * Poor way to test classes!
 *
 * @param args
 */
public static void main(String[] args) {
  // three locales to use for the test
  Locale locales[] = {
     Locale.ITALY,
     Locale.UK,
     Locale.JAPAN
  };

  // formatting a simple number into a string
  for (Locale currLocale : locales) {
    LOGGER.info(
       currLocale.getCountry()+"\t"+
       NumberFormat
         .getNumberInstance(currLocale)
         .format(-1234.56) );
  }

  // parsing the string into a number
  for (Locale currLocale : locales) {
    try {
      Number number =
      NumberFormat
        .getNumberInstance(currLocale)
        .parse("-1.234,56");

      if (number instanceof Long) {
```

```
        LOGGER.info("The number:"+number.longValue()+" is a
long");
      } else {
        LOGGER.info("The number:"+number.doubleValue()+" is a
double");
      }

    } catch (ParseException e) {
      LOGGER.warn("Parsing exception");
    }
  }
 }
}
```

Listato 2. *Semplice esempio di uso di Locale.*

Di seguito è mostrato l'esempio prodotto dall'esecuzione del precedente listato

```
IT -1.234,56
GB -1,234.56
JP -1,234.56
The number:-1234.56 is a double
The number:-1.234 is a double
The number:-1.234 is a double
```

La prima sezione del programma si occupa di formattare un numero. E qui la situazione sembrerebbe abbastanza innocua: i diversi numeri sono formattati secondo lo standard della nazione impostata. Tuttavia, l'Internazionalizzazione deve essere utilizzata molto attentamente. Si provi ad immaginare a cosa potrebbe accadere se un Trader in quel di Londra per sbaglio si trovasse ad operare con un locale in italiano! Pensando di scrivere dei valori decimali si troverebbe a scrivere delle migliaia! Quindi una regola d'oro consiste nel mostrare sempre chiaramente il locale che si sta utilizzando.

Nella seconda parte del codice, avviene l'operazione opposta, una stringa è convertita nel corrispondente numero. In questo caso è evidentissimo l'impatto del locale: lo stesso numero può variare dalle migliaia al punto decimale.

Aggiornamenti API UI

Java SE 7 include anche una serie di aggiornamenti della UI. Vediamoli nei prossimi paragrafi.

Nimbus

Con il JDK 7, il look-and-feel di **Swing Nimbus** (cfr. [Nimbus, 2012]), introdotto nel JDK 6u10 come sostituzione per il vecchio **Metal LoF**, è stato spostato dalle estensioni proprietarie di "Oracle" (**com.sun.java.swing**) alla libreria standard Swing (**javax.swing**) facendolo diventare un vero e proprio cittadino di prima classe della libreria **Swing**. Sebbene presenti tutta una serie di vantaggi, inclusi migliore look&feel cross-platform e implementazione più scalabile della parte Java 2D, non è stato impostato come default proprio per non creare problemi alle applicazioni esistenti.

JLayer

Con Java SE 7 è stata introdotta una nuova classe Swing: **JLayer**. Si tratta di un **decoratore universale** per componenti Swing che permette sia di realizzare vari sofisticati effetti di paint, sia di ricevere le notifiche di tutti gli **AWTEvents** generati al suo interno. **JLayer** delega la gestione del disegno e degli eventi input a un oggetto **LayerUI** (**javax.swing.plaf.LayerUI<V extends Component>**), il quale esegue la decorazione vera e propria. Il paint personalizzato implementato nel **LayerUI** e la notifica eventi esegue il lavoro per il **JLayer** stesso e tutte le sue sotto-componenti. Questa combinazione consente di arricchire i componenti esistenti con l'aggiunta di nuove funzionalità avanzate come la chiusura temporanea di una gerarchia, consigli dati per i componenti composti, un maggiore scorrimento del mouse e così via.

JLayer è una soluzione valida quando si ha bisogno di eseguire il painting customizzato per componente complesso formato da sotto-componenti e per intercettare eventi per i vari sotto-componenti.

Standardizzazione delle finestre traslucenti

Java 7 consente di **definire il grado di trasparenza** di una finestra, ad esempio un JFrame. Una finestra 90% traslucida è 10% opaco (quindi non si vede per il 10%). Il grado di trasparenza per una finestra può essere definita utilizzando il valore alfa della componente di colore di un pixel. È possibile definire il valore alfa di un colore utilizzando i costruttori della classe colore come illustrato di seguito.

```
Color (int red, int green, int blue, int alfa)
```

```
Color (float red, float verde, blu float, float alpha)
```

Il valore per l'argomento alpha viene specificato tra **0** e **255**, quando i componenti del colore sono specificati come valori interi. Quando invece gli argomenti sono di tipo **float**, il valore è compreso tra **0,0** e **1,0**. Il valore alpha di 0 o 0,0 significa "modo trasparente" (si vede attraverso). Il valore di alfa 255 o 1,0 è "opaco" (quindi non si vede attraverso).

Java 7 supporta le seguenti tre forme di trasparenza di una finestra, rappresentate dalle seguenti tre costanti incluse nella enumeration **java.awt.GraphicsDevice.WindowTranslucency**.

- **PERPIXEL_TRANSPARENT**: in questa forma di traslucidità, un pixel in una finestra è o opaco o trasparente: il valore alfa per un pixel è 0,0 o 1,0.
- **TRANSLUCENT**: in questa forma di traslucidità, tutti i pixel in una finestra hanno la stessa traslucenza, che può essere definita da un valore di alfa tra 0,0 e 1,0.
- **PERPIXEL_TRANSLUCENT**: in questa forma di trasparenza, ogni pixel in una finestra può avere un proprio valore alpha compreso tra 0,0 e 1,0. Questo modulo permette di definire la traslucenza in una finestra su una base per pixel.

Il listato seguente mostra l'utilizzo di alcune feature menzionate poc'anzi. In particolare, la classe ShapeWindow si occupa di disegnare una finestra poligonale

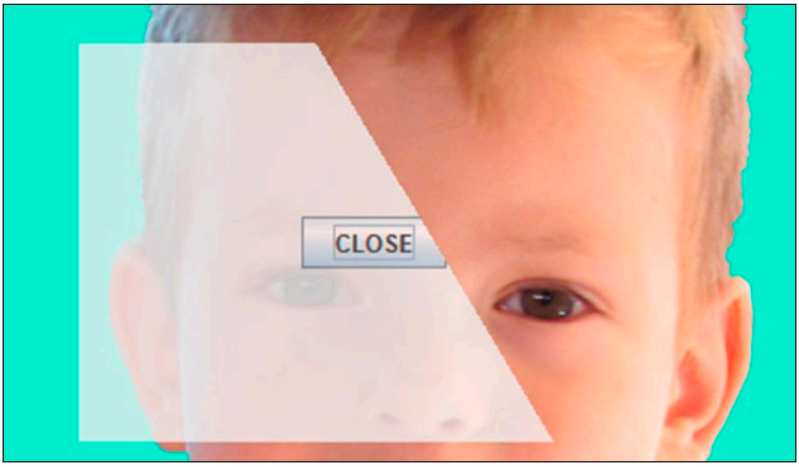


Figura 1. *Effetto dell'esecuzione della classe su una fotografia.*

con sfondo traslucente, nella quale è inserito un tasto che permette di chiudere il programma. Per evidenziare questa funzionalità di traslucenza (impostata ad un valori pari all'80%), il programma è stato eseguito su uno sfondo con una fotografia come mostrato in figura 1.

Il codice è sufficientemente documentato tanto che la sua comprensione dovrebbe essere immediata.

```
public class ShapedWindow extends JFrame {

  // ----------------- CONSTANTS SECTION -----------------

  /** logger */
  private static final Logger LOGGER =
     Logger.getLogger(ShapedWindow.class);

  // ----------------- ATTRIBUTES SECTION ----------------

  // ----------------- METHODS SECTION -------------------

  /**
   * Constructor method
   */
  public ShapedWindow() {

    super("ShapedWindow");
    LOGGER.info("Designing the window...");

    setLayout(new GridBagLayout());

    // It is best practice to set the window's shape in
    // the componentResized method. Then, if the window
    // changes size, the shape will be correctly recalculated.
    addComponentListener(new ComponentAdapter() {

      // give the window a polygon shape.
      // If the window is resized, the shape is recalculated here.
      @Override
      public void componentResized(ComponentEvent e) {
```

```
        int[] polygonXs =
          { -getWidth(),  0, getWidth(),  0};
        int[] polygonYs =
          { getWidth()/2, getHeight(), getWidth(), -getHeight()};

        setShape(
           new Polygon(polygonXs, polygonYs, polygonXs.length) );
      }
    });

    setUndecorated(true);
    setSize(300, 200);
    setLocationRelativeTo(null);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // create a close button
    JButton closeButton = new JButton("CLOSE");
    closeButton.addActionListener(new ActionListener() {
      /**
       * Receives button events
       * @param event button events
       */
      public void actionPerformed(ActionEvent event) {
          System.exit(0);
      }
    });

    add(closeButton);
  }


  /**
   * Main method
   * @param args  command line parameters
   */
  public static void main(String[] args) {
    // Determine what the GraphicsDevice can support.
    GraphicsEnvironment ge =
```

```
        GraphicsEnvironment.getLocalGraphicsEnvironment();

    GraphicsDevice gd =
        ge.getDefaultScreenDevice();

    final boolean isTranslucencySupported =
        gd.isWindowTranslucencySupported(TRANSLUCENT);

    // if shaped windows aren't supported, terminate with -1.
    if (!gd.isWindowTranslucencySupported(PERPIXEL_TRANSPARENT)) {
      LOGGER.info("Shaped windows are not supported");
      System.exit(-1);
    }
    LOGGER.info("Shaped windows are supported.");

    // If translucent windows aren't supported,
    // create an opaque window.
    if (!isTranslucencySupported) {
      LOGGER.info(
          "Translucency is not supported,"+
          " creating an opaque window");
    } else {
      LOGGER.info("Translucency is supported.");
    }

    // Create the GUI on the event-dispatching thread
    SwingUtilities.invokeLater(new Runnable() {

      @Override
      public void run() {
        ShapedWindow sw = new ShapedWindow();

        // Set the window to 80% translucency, if supported.
        if (isTranslucencySupported) {
          sw.setOpacity(0.8f);
        }

        // Display the window.
```

```
        sw.setVisible(true);
      }
    });
  }

} // --- end of class ---
```

Listato 3. *Esempio di alcune nuove feature Swing (come traslucent e setShape).*

OpenType/CFF font

Java SE 7 introduce il supporto per il **Compact Font Format** OpenType/CFF (font dall'estensione **.otf**) e TrueType (font dall'estensione **.ttf**): si tratta di due formati di font moderni disponibili per l'uso su desktop. Pur essendo formati distinti, OT/CFF e TTF in realtà hanno molte similitudini e le differenze risiedono principalmente nei differenti formati outline (di struttura) e nei diversi approcci impiegati per eseguire il rastering dei contorni. In particolare, il font OT/CFF utilizza una strategia basata sulla curve cubiche di Bezier mentre in caratteri TTF utilizzano i percorsi basati sulle curve quadratiche di Beziers.

NumericShaper

La classe **java.awt.font.NumericShaper** è utilizzata per convertire cifre in formato europeo (Latin-1) in altre rappresentazioni Unicode. La strategia consiste nel presentare numeri secondo i formati locali gestendoli internamente in formato europeo (Latin-1). Questa classe non è in grado di interpretare il carattere numerico obsoleto (U+206 E). Le istanze di tipo **NumericShaper** sono generalmente applicate come attributi al testo con l'attributo **NUMERIC_SHAPING** della classe **TextAttribute**.

XRender per Java 2D

Altro elemento da menzionare è l'introduzione di una **pipeline grafica XRender** per Java 2D, atta a migliorare la gestione delle funzionalità specifiche delle GPU moderne (X11-based desktop).

JDBC 4.1

Java SE 7 include la nuova versione JDBC (4.1, cfr. [JDBC 4.1, 2012]) con le seguenti caratteristiche:

- la possibilità di utilizzare le risorse **Connection**, **ResultSet**, e **Statement** in costrutti **try-with-resource** introdotti con il progetto **Coin**;

- l'introduzione dell'interfaccia **RowSetFactory** e della classe **RowSetProvider**, che consentono di creare tutti i tipi di **RowSet** supportati dal driver JDBC; in particolare, l'interfaccia **RowSetFactory** contiene i seguenti metodi che permettono di creare i corrispondenti tipi di **RowSet**: **createCachedRowSet**, **createFilteredRowSet**, **createJdbcRowSet**, **createJoinRowSet** e **createWebRowSet**.

Elliptic Curve Cryptography (ECC)

Con Java SE 7 è stata introdotta l'implementazione dell'algoritmo **Elliptic Curve Cryptography** (**ECC**, crittografia basata sulle curve ellittiche), In particolare, Sun fornisce SunEC come provider dei servizi ECC. Si tratta di un approccio di crittografia per chiavi pubbliche basato sulla struttura algebrica delle curve ellittiche su campo finito. ECC è ideale per l'utilizzo in ambienti con scarse risorse come pager (cercapersone), PDA, telefoni cellulari e smart card. Rispetto ai tradizionali sistemi di crittografia come RSA, ECC offre un equivalente livello di sicurezza con ridotte dimensioni delle chiavi, in che si traduce in calcoli più efficienti e minore consumo energetico, di risorse e di risparmio di larghezza di banda. L'implementazione Java è stata denominata **JECC**. L'utilizzo di questo algoritmo per la crittografia fu suggerito in maniera indipendente sia da Neal Koblitz sia da Victor S. Miller nel 1985.

Conclusioni

Con questo capitolo abbiamo terminato la presentazione delle nuove feature introdotte con Java SE7. In particolare, abbiamo presentato tutta una serie di migliorie relative al passaggio alla versione **6 di Unicode**, al supporto estensibile per il codice standard delle valute, le variazioni alle impostazioni della localizzazione (**Local**), le nuove feature della **GUI** (come Nimbus, JLayer, Translucent Window), la nuova versione JDBC 4.1 e il supporto per l'algoritmo di crittografazione ECC.

Capitolo 6
Aspettando la release 8

Introduzione

L'obiettivo di questo lungo capitolo è presentare una overview di Java SE 8: major release programmata inizialmente per l'estate 2013. Considerate le recenti discussioni e problematiche presenti all'interno della comunità Java, è ormai sicuro che tale scadenza subirà ulteriori ritardi (al momento in cui viene redatto questo capitolo si parla dell'autunno 2013, ma è tutto da confermare). Inoltre, la storia di Java SE 7 ci insegna che probabilmente diverse caratteristiche programmate per Java SE 8, verranno ulteriormente posticipate alle versione successiva (Java SE9).

Secondo i piani iniziali, le principali innovazioni da introdurre con Java SE 8 dovevano essere:

- molteplici proposte del progetto **Coin** (Moneta) escluse da Java SE 7;
- realizzazione del progetto di modularizzazione di Java, **Jigsaw** (puzzle, iniziato originariamente da Sun Microsystem nel 2008).
- introduzione delle **Lambda Expression**, espressioni Lambda ossia programmazione funzionale.

Per quanto concerne il progetto Jigsaw, come già accaduto con la versione precedente, le cose non stanno procedendo come atteso e quindi si va quasi sicuramente verso un notevole ritardo o addirittura un ulteriore discope, che qualche imbarazzo dovrebbe creare in casa Oracle.

Il progetto, Lambda, come lecito attendersi, ha creato e sta creando grande interesse intorno alla versione Java SE 8. Nel web è possibile trovare molti commenti tra persone soddisfatte dei cambiamenti e persone che invece avrebbero voluto un supporto decisamente più spinto della programmazione funzionale (da molti considerata il vero futuro di Java), più orientato a soluzioni come quelle offerte da Scala. Sicuramente si tratta di un cambiamento importante che influenza diverse parti di Java: linguaggio, librerie e ambiente. In questo capitolo riportiamo una presentazione abbastanza dettagliata delle espressioni Lambda, senza dimenticarci di evidenziarne pregi e inevitabili lacune.

Prima di procedere, riportiamo di seguito la più recente **roadmap** rilasciata da Oracle ("ci troviamo in periodo di transizione... Come sempre d'altronde."), che seguendo best practice della disciplina del project management è organizzata in iterazioni, ognuna della quale ha il compito di implementare un ben definito set di feature ed il cui termine previsto è cadenzato da milestone ben definite:

```
26 Aprile 2012     M1 (milestone "interna")
14 Giugno 2012     M2 (milestone "interna")
02 Agosto 2012     M3 (milestone "interna")
13 Settembre 2012  M4 (milestone "interna")
29 Novembre 2012   M5 (milestone "interna")
31 Gennaio 2013    M6 Feature complete
21 Febbraio 2013   M7 Developer preview
05 Luglio 2013     M8 Final release candidate
09 Settembre 2013     Rilascio
```

Per finire vale la pena ricordare che al momento in cui viene redatto questo capitolo, Java SE 8 non è ancora definitivo, quindi alcune parti potrebbero subire dei cambiamenti. Tuttavia, questi non dovrebbero riguardare il core.

Coin 2

Come illustrato nel capitolo 1 "Java SE7 in breve" la comunità Java al fine di evitare ulteriori ritardi nel rilascio di Java SE 7, ha votato per l'attuazione del famoso "Piano B", il quale, oltre a posticipare la consegna delle Lamba Expression e del progetto Jigsaw ha imposto una riduzione delle nuove caratteristiche incluse del progetto Coin da far confluire nella release Java SE 7. Alcune delle rimanenti feature verranno fatte confluire nella release Java SE 8. Di seguito è riportata la lista delle rimanenti proposte di modifica del linguaggio Java valutate all'interno del progetto Coin (cfr. [Darcy, 2009]), con tipo di proposta, propositore e data .

```
Block Expressions for Java
Neal Gafter 28/02/2009

Improved Exception Handling for Java
Neal Gafter 28/02/2009

Improved Wildcard Syntax for Java
Neal Gafter 28/02/2009
```

Multiline strings/String literals
Ruslan Shevchenko 01/03/2009

Elvis and Other Null-Safe Operators
Stephen Colebourne 01/03/2009

Import Aliases for Classes and Static Methods
Phil Varner 03/03/2009

Lightweight Properties
David Goodenough 03/03/2009

Static Methods in Interfaces
Reinier Zwitserloot 04/03/2009

Multiple switch expressions and case ranges
Pinku Surana 05/03/2009

Fold keyword
Gabriel Belingueres 05/03/2009

Extend Scope of Imports to Include Package Annotations
Bruce Chapman 09/03/2009

Enhanced String constructs for Java
Felix Frost 09/03/2009

Method and Field Literals
Jesse Wilson 11/03/2009

Embedded Expressions for String Statements
Stefan Schulz 15/03/2009

Access to Generic Type Parameters at Compile-Time
David Walend 17/03/2009

Named method parameters
Paul Martin 21/03/2009

Enhanced for each loop iteration control
Stephen Colebourne 21/03/2009

Simplified syntax for dealing with parameterized types
James Lowden 23/03/2009

Compiletime information access
Ruslan Shevchenko 24/03/2009

@Unloadable Class definition
Daniel Cheng 24/03/2009

Large arrays
James Lowden 24/03/2009

Auto-assignment Parameters
Mark Mahieu 25/03/2009

Unchecked Exceptions as Subclasses of Checked Exceptions
Alan Snyder 25/03/2009

Byte and Short Integer Literal Suffixes
Bruce Chapman 25/03/2009

Unsigned Integer Widening Operator
Bruce Chapman 25/03/2009

Return ‘this’
Marek Kozie 25/03/2009

Narrow Hexadecimal and Binary Integer Literals
Bruce Chapman 25/03/2009

Improved Support for Optional Object Behaviors at Runtime
Alan Snyder 26/03/2009

‘final’ without explicit type
Marek Kozie 26/03/2009

Language Escape Operator
Bruce Chapman 26/03/2009

'forget' keyword (beta)
Marek Kozie 27/03/2009

Using Ordinary Syntax for Reflective Method Invocations
Alan Snyder 27/03/2009

Type inference for variable definition/initialization using the 'auto' keyword
Tim Lebedkov 28/03/2009

Simplified StringBuffer/StringBuilder syntax
Derek Foster 28/03/2009

Simple Operator Overloading
Ruslan Shevchenko 29/03/2009

Conditional Statement
Matt Mastracci 29/03/2009

Enhanced while statement
Marek Kozie 29/03/2009

Abstract Enums
Derek Foster 29/03/2009

Open and close brace optional for single statement try, catch, finally, method declaration
Glenn A.P. Marshall 30/03/2009

Concise Collection Initialization Syntax
Shams Mahmood 30/03/2009

Allow the class literal of the surrounding class to be referenced with the 'class' keyword
Peter Runge 30/03/2009

```
   Generic Specification by Method
   Ron Thomas 30/03/2009

   checked exception handling enhancement
   evildeathmath at yahoo.com 30/03/2009

   Rethrows Clause
   Mark Mahieu 30/03/2009

   Elided type specification for initialized final value declarations
   Mike Elliott 30/03/2009

   Templated Construction Expressions (i.e. Expressions
Embedded in Strings)
   John Rose 30/03/2009

   Sameness operators/Equivalence Operators
   Derek Foster 31/03/2009

   Glue classes based on Extension Methods
   Marek Kozie 31/03/2009

   Accepting a subclass as an element type in a for loop
   Jean-Louis Ardoint 31/03/2009

   @OverrideAll annotation
   Gabriel Belingueres 31/03/2009

   Improved declaration using final keyword of internal method variables
and for-loop statement
   Andrej Navodnik 31/03/2009

   Fast and accurate measurement of execution time
   Angelo Borsotti 13/04/2009

   Indexing access syntax for Lists and Maps
   Derek Foster 22/04/2009
```

```
Allow a single-parameter method to be used as an infix operator
Elias Ross 23/04/2009

Underscores in Numbers (Version 2)
Derek Foster 01/05/2009

Enhanced for each loop iteration control
Derek Foster 01/05/2009

Improved Support for Optional Object Behaviors atRuntime
Stefan Schulz 12/05/2009

Byte and Short Integer Literal Suffixes
Derek Foster 21/05/2009

Concise declaration of JavaBeans properties
Jan Kroken 25/06/2009
```

Le proposte di cambiamento maggiormente discusse

Allo stato attuale non è ancora chiaro ai non addetti ai lavori (e forse non solo a loro...) quante e quali di queste feature confluiranno in Java SE 8. Tuttavia, dall'analisi delle discussioni all'interno del progetto Coin, è possibile notare intense discussioni relative alle seguenti proposte:

- Multiline strings/String literals
- Elvis and Other Null-Safe Operators
- Static Methods in Interfaces
- Large arrays;
- Concise Collection Initialization Syntax
- Sameness operators/Equivalence Operators
- Type annotations

Anche se alcune di queste feature non verranno effettivamente incluse, rimane tuttavia interessante analizzare elementi di discussione e proposte relativi a cambiamenti di sintassi e semantica del linguaggio di programmazione . La maggior parte di questi sono nati da esigenze concrete e che quindi la risoluzione dovrebbe veramente portare ad un "miglioramento della vita del programmatore". Inoltre è importante notare che oltre ad alcune di queste proposte, Java SE 8 dovrà includere una serie di cambiamenti inerenti sia il linguaggio,

sia le API necessari per un adeguato supporto delle Lambda Expression discusse nell'apposito paragrafo.

È interessante comunque riportare una descrizione delle proposte più in auge.

Multiline strings/String literals

La richiesta consiste nell'aggiungere la possibilità di avere stringhe multi-linea in Java al fine di rendere il codice più elegante e leggibile.

```
StringBuilder sb = new StringBuilder();
sb.append( """select a from Area a, CountryCodes cc
   where
      cc.isoCode='IT'
     and
      a.owner = cc.country
 """);
if (question.getAreaName()!= null) {
  sb.append( """and
      a.name like ?
   """);
   sqlParams.setString(++i,question.getAreaName());
}
```

Listato 1. *Esempio di stringa multi-linea.*

Secondo questa proposta, il compilatore dovrebbe inserire ogni riga in un apposito append e terminarlo con un carattere nuova linea (da notare che \n non è completamente platform independent). Come riportato nel listato seguente.

```
StringBuilder sb = new StringBuilder();
sb.append("select a from Area a, CountryCodes cc\n");
sb.append("where cc.isoCode='IT'\n");
sb.append("and a.owner=cc.country'\n");
if (question.getAreaName()!= null) {
  sb.append("and a.name like ?");
  sqlParams.setString(++i,question.getAreaName());
}
```

Listato 2. *Gestione del multilinea.*

Elvis and Other Null-Safe Operators

L'idea alla base di questa proposta è di semplificare il controllo e la gestione dei puntatori **null**. In particolare, l'idea consiste nell'introdurre un nuovo operatore, **Elvis** appunto, che si materializza attraverso un punto esclamativo in grado di continuare la navigazione degli oggetti solo qualora questi siano correttamente inizializzati (non siano null).

Il seguente listato chiarisce la proposta.

```
public String getPersonPostcode(Person person) {
  String postcode = null;
  if (person != null) {
    Address address = person.getAddress();
    if (address != null) {
      person = address.getPostcode();
    }
  }
  return postcode
}

//Equivarrebbe a:
public String getPersonPostcode(Person person) {
  return person?.getAddress()?.getPostcode();
}
```

Listato 3. *Esempio di utilizzo dell'operatore Elvis.*

Static Methods in Interfaces

Il nome di questa feature dovrebbe essere auto esplicativa: il linguaggio Java attualmente non consente di definire metodi statici all'interno delle interfacce, quindi l'obiettivo di questa proposta è di rimuovere tale limitazione. In particolare, si richiede la possibilità di definire l'implementazione di metodi statici all'interno delle interfacce rimuovendo la classica implicazione dei metodi definiti nelle interfacce: loro definizione nelle classi che implementano l'interfaccia. Ciò, oltre ad essere utile alla programmazione funzionale (come descritto nell'apposito paragrafo) dovrebbe agevolare l'implementazione di metodi di utilità all'interno di una serie di interfacce. Alcuni esempi sono la possibilità di definire l'implementazione del metodo **sort** direttamente nell'interfaccia **java.util.List** e **unmodifiableMap** nell'interfaccia **java.util.Map**. La soluzione utilizzata

attualmente consiste nel ricorrere ad opportune classi separate che contengono questi metodi di utilità come java.util.Collections. Questa soluzione ovviamente non rappresenta un elegante disegno OO e riduce la potenza delle API.

Large arrays

Allo stato attuale, gli Array in Java sono accessibili tramite interi a 32-bit. La logica conseguenza è che la dimensione massima teorica è di 2^31 = 2.147.483.647 elementi. Si tratta di un limite che nella stragrande maggioranza dei casi non è minimamente avvertito, tuttavia esistono delle applicazioni che invece devono fare i conti con grandi insiemi di dati sequenziali da mantenere in memoria. In queste situazioni risulterebbe decisamente vantaggioso poter disporre di array indicizzati con indici a 64 bit. Una soluzione immediata potrebbe essere di estendere questa soluzione a tutti i casi. Tuttavia come mostrato nella serie di articoli apparsi su MokaByte (cfr. [Vetti Tagliati, 2010]) ciò potrebbe portare ad una riduzione delle performance. Pertanto inizialmente si era pensato ad un cambiamento di sintassi al fine di consentire al programmatore di dichiarare esplicitamente i casi in cui è veramente necessario questo supporto. La proposta finale invece prevede di evidenziare la richiesta di array di grandi dimensioni in modo assolutamente naturale e standard aggiungendo la dimensione esplicita nell'array e quindi demandando al compilatore il compito di desumere se la dimensione richieda o meno 64 bit di indirizzamento.

```
int [] largeArray = new int [3000000000L];
largeArray[2999999999L] = 500;
```

Listato 4. *Esempio di large array.*

Concise Collection Initialization Syntax

Le classi del framework delle collezioni sono indubbiamente tra le più frequentemente utilizzate nella scrittura di programmi in Java. Come lecito attendersi, presentano importanti analogie con array e matrici. Tuttavia, a differenza degli array, le collezioni non permettono un'inizializzazione semplificata: è necessario ricorrere a particolari blocchi statici e/o a classi annidate anonime. Questa proposta mira a fornire un supporto aggiuntivo del linguaggio Java che consenta una concisa inizializzazione delle collezioni a tempo di compilazione.

Il seguente listato mostra alcuni esempi della feature proposta:

```
public class ConciseCollectionInit {
```

```
  public static void main(String[] arguments) {
    List<String> myList
      = new LinkedList<String>()
          [“Hello”, “world”, “!” ];

    Map<Integer, String> myMap =
      new HashMap<Integer, String>()
          {} [1:”a”, 2:”b”, ];

  }
}
```

Listato 5. *Inizializzazione concisa delle classi del framework Collection.*

Sameness operators/Equivalence Operators

Nell’implementazione delle classi Java è molto frequente eseguire l’override del metodo equals (Object.equals) e di implementare l’interfaccia **Comparable <T>** al fine di implementare dei servizi che permettono di verificare se due istanze sono o meno equivalenti e di ordinare insiemi di oggetti. Il problema è che la sintassi per l’implementazione di questi metodi in Java è poco elegante e spesso ostica. Questa proposta mira a definire un nuova sintassi (unita ad apposita semantica) per migliorare il codice da scrivere per questi casi. Molto brevemente la nuova sintassi è descritta di seguito:

```
a $$ b “same as”:
(a==null ? b==null : a.equals(b), or a == b for primitive types.

a !$ b “not same as”:
a==null ? b!=null : !a.equals(b), or a != b for primitive types.

a >$ b “greater than or same”:
a.compareTo(b) >= 0, or a >= b for primitive types.

a <$ b “less than or same”:
a.compareTo(b) <= 0, or a <= b for primitive types.

Oltre che permettere l’overloading degli operatori come segue:
a < b  a.compareTo(b) < 0, or a < b for primitive types.
```

```
a > b  a.compareTo(b) > 0, or a > b for primitive types.
```

Listato 6. *La nuova sintassi proposta.*

Questa proposta presenta diversi punti di contatto con le espressioni Lambda e soprattutto una soluzione divergente. Pertanto è interessante se si punterà esclusivamente sulle espressioni Lambda o se si tenterà comunque di portare avanti entrambi gli approcci.

Type annotations

Il supporto delle annotazioni è stata una delle novità introdotte in Java 1.5 (aka J2SE 5.0) di maggiore successo che tra l'altro ha posto le basi per una mini rivoluzione nel mondo degli EJB container e application server. Tuttavia, il loro utilizzo a tutt'ora (Java SE 7) è limitato ai parametri dei metodi, alla dichiarazione di package, classi, metodi, field e variabili locali. La proposta type annotations lancia l'idea di un utilizzo più generale e quindi ancora più utile delle annotazioni. In particolare, l'idea consiste nel poter annotare la dichiarazione di tipo **al fine di migliorare la qualità del codice e per prevenire/individuare errori a tempo di compilazione che altresì emergerebbero a tempo di esecuzione**. Qui di seguito è mostrato un esempio con un nuovo insieme di annotazioni:

```
@Untainted String query;
List<@NonNull String> strings;
myGraph = (@Immutable Graph) tmpGraph;
class UnmodifiableList<T>
implements @Readonly List<@Readonly T> {}
```

Lisato 7. *Esempio di utilizzo delle type annotations*

La proposta tuttavia si limita a **specificare la sintassi e non la semantica** evitando così la necessità di dover modificare il compilatore, e l'ambiente di esecuzione. Ciò equivale a dire che le annotazioni dei tipi risulterebbero dei **controlli "pluggable"** ad uso e consumo di opportuni **pre-processori**.

Jigsaw

Lo scopo del progetto Jigsaw ("puzzle") è progettare e realizzare un sistema modulare standard per la piattaforma Java da applicare sia alla stessa piattaforma,

sia al JDK. Si tratta di realizzare un sistema che sia una **sorta di mix tra Maven** (cfr. [Vetti Tagliati, 2007]) **e OSGi** (l'acronimo deriva originariamente da **O**pen **S**ervices **G**ateway **i**nitiative, tuttavia tale definizione è stata deprecata, cfr. [OSGi, 2012]). Da una parte Maven da solo non è sufficiente. Maven è un tool per la gestione e comprensione dei progetti e quindi è uno strumento alquanto valido per organizzare progetti, gestirne il ciclo di vita, le relative informazioni, e così via, ma, per sua natura, si limita agli aspetti relativi a tempo di costruzione (**build-time**). In altre parole, **Maven non si occupa di aspetti di modularizzazione relativi al tempo di esecuzione**.

Dall'altro lato, anche **OSGi**, che logicamente può essere collocato all'altro estremo del campo di competenza, considerato da solo **non è sufficiente**. Il framework OSGi è un sistema modulare e una piattaforma di servizi per il linguaggio di programmazione Java che implementa un modello a componenti completo e dinamico. **OSGi**, pertanto, si occupa degli aspetti di **modularizzazione a tempo di esecuzione** (per essere precisi interviene a partire dal packaging includendo il deployment e l'esecuzione) e molto meno degli aspetti di organizzazione del progetto.

La logica conseguenza è che, almeno da un punto di vista teorico, **Jigsaw** dovrebbe essere un'opportuna **combinazione tra Maven e OSGi**, combinazione, che ad onore del vero, è già una sorta di realtà in diverse organizzazioni. Il limite di questa strategia è che si tratta di una **soluzione creata a partire dalla piattaforma Java**, mentre una vera innovazione dovrebbe richiedere cambiamenti radicali a partire dalla piattaforma stessa. Si tratta di un compito complicato considerando che la base di partenza è un **code-base** che si è evoluto negli anni secondo un **modello abbastanza monolitico e non modulare** come necessario per dar luogo ad una vera modularizzazione dell'ambiente. Una conseguenza di ciò, per esempio, è che il codice JDK sia profondamente interconnesso sia al livello delle API sia ai vari livelli di implementazione sottostante. Pertanto, una completa ed accurata implementazione del progetto richiede un lavoro propedeutico atto a **eliminare** o quanto meno minimizzare **le varie interdipendenze**. Ciò è necessario per far sì che sia la piattaforma, sia le sue implementazioni possano effettivamente diventare un insieme coerente di moduli interdipendenti.

Stato del progetto

Il modo molto conveniente per illustrare l'evoluzione del progetto Jigsaw consiste nel riportare la seguente sintesi dei commenti inseriti da Mark Reinhold, Chief Architect del Java Platform Group presso Oracle, nel suo blog (cfr. [Reinhold, 2012]).

La realizzazione del progetto Jigsaw era prevista per Java 7 e poi per Java 8. Al fine di raggiungere quest'ultimo obiettivo, il lavoro sulle caratteristiche principali dovrebbe essere terminato entro maggio 2013, ciò per consentire la preparazione di una versione finale per settembre. Nonostante gli importanti progressi, permangono tuttavia una serie di problemi tecnici di una certa importanza. La mancanza di tempo, dovuta all'appressarsi della dead-line inoltre complica la valutazione generale, la revisione e il feedback di un cambiamento così radicali. Constatati questi problemi, la proposta di Reinhold consiste nel **rinviare ulteriormente** il progetto Jigsaw alla successiva release, **Java SE 9**. Inoltre, al fine di aumentare la prevedibilità di tutte le future release Java SE, Reinhold propone di stabilire il ciclo di rilascio delle nuove versioni su un arco di tempo di **due anni**.

Sempre secondo Reinhold, nonostante i progressi, permangono alcune importanti sfide tecniche da sormontare, tra cui la completa modularizzazione della piattaforma Java SE e del JDK, mantenendo, al tempo stesso, la compatibilità con il codice esistente. Si tratta ovviamente di un compito estremamente delicato che richiede modifiche ben ponderate in ogni parte, a partire dalle specifiche per finire all'implementazione e test. Inoltre, è necessario progettare e prototipare una strategia al sostegno di container come IDE, application server Java EE container, e applet container, i quali tendono a sfruttare massicciamente binding dinamici e reflection. Lo stato del progetto lascia pensare che il team abbia una certa confidenza di riuscire a risolvere i vari problemi, tuttavia è innegabile che si sia accumulato un certo ritardo tanto da compromettere l'attuale deadline dell'intera release di Java SE 8. L'introduzione di una piattaforma veramente modulare è un passaggio importante che, alla fine, influenza l'intero ecosistema Java. Questa ha la potenzialità di cambiare il modo in cui la piattaforma Java stessa viene distribuita, il modo in cui gli sviluppatori creano e distribuiranno librerie, framework, tool e applicazioni. Pertanto è di fondamentale importanza avere tempo a sufficienza per la revisione generale, eseguire test ed analizzare il feedback ricevuto da tutti i segmenti della comunità Java relativi sia alla progettazione, sia sull'implementazione.

A questo punto, quindi, le opzioni disponibili sono "tristemente" familiari dall'esperienza di Java SE 7:

- posporre a Java SE 9 la consegna del progetto Jigsaw al fine di non ritardare la consegna di Java SE 8;
- attendere la conclusione dei lavori del progetto Jigsaw, facendo slittare la consegna di della versione Java SE 8 verso la metà del 2014.

Questo nuovo, ulteriore "piano B" (al momento in cui viene scritto questo capitolo) è oggetto di accese discussioni all'interno della comunità Java e, come da

protocollo, **la decisione finale è affidata al (JSR 337) Expert Group**. Tuttavia la domanda che ricorre nella comunità Java è se questa ennesima posticipazione non mini alla base la futura accettazione di una soluzione come Jigsaw. Si tratta di un dubbio assolutamente legittimo considerando anche i grandi progressi effettuati da OSGi.

Lambda Expression

La **programmazione funzionale** è un importante paradigma che ha ispirato diversi linguaggi di programmazione, basti citare LISP e Scala, che hanno concorso e concorrono a fare la storia dell'Informatica. Una piena comprensione delle espressioni Lambda, come lecito attendersi, richiede la conoscenza di alcuni principi base di programmazione funzionale. Al tal fine è stata redatta l'apposita appendice D, "Stile funzionale in un chicco di grano" demandando al lettore la decisione di leggerla o meno, con il monito che la conoscenza delle poche nozioni riportate è propedeutica alla piena comprensione dei paragrafi successivi, dedicati alla presentazione del progetto Lambda. Particolare attenzione è stata attribuita alle principali feature che verranno introdotte al fine di integrare in Java una serie di elementi di programmazioni funzionale.

Tutti coloro che si attendono un cambiamento radicale della piattaforma Java in grado di colmare il gap con linguaggi come Scala verranno delusi: l'**evoluzione del linguaggio Java non è assolutamente un esercizio agevole**, ci sono molti vincoli da tenere presente come la "sacra" compatibilità con le versioni passate, l'esistenza di un ecosistema maturo che include sofisticati compilatori, JVM per le diverse piattaforme, application server, esteso set di librerie, e così via. Pertanto, a partire da J2SE 5.0 con l'introduzione di feature come generics ed il ricorso alla strategia dell'erasure, la comunità Java ha iniziato a capire intimamente come l'evoluzione controllata di Java debba necessariamente risolversi nel trovare accettabili compromessi tra l'implementare di feature in modo incontaminato e la compatibilità con le versioni precedenti.

Il progetto Lambda (JSR 355)

La storia del progetto Lambda (chiamato anche informalmente "closures" e "anonymous methods"), come riportato già nel capitolo 1 ("Java SE7 in breve"), è stata un po' travagliata: proposto inizialmente per essere rilasciato con Java SE 7, dopo diversi importanti ritardi, dovuti ad accese discussioni all'interno della comunità, è stato alla fine posticipato a Java SE 8, secondo i dettami del famoso piano B. Come riportato di seguito, si tratta di un'importante feature in grado di

trasformare profondamente sia il linguaggio Java, sia il JDK, sia tutti i framework e librerie esistenti.

Il progetto è **guidato Brian Goetz**, Java Language Architect alla Oracle Corporation, in precedenza Senior Engineer alla Sun Microsystem, autore di innumerevoli pubblicazioni tecniche di grande interesse e coautore di uno dei migliori libri sul multi-threading in java ([Goetz et al, 2006]).

L'introduzione del supporto del paradigma funzionale in Java è un esercizio delicato e complicato allo stesso tempo. Una vasta percentuale di sostenitori di Java sono entrati in contatto, più o meno approfonditamente, con linguaggi quali Clojure e Scala completamente basati sul paradigma funzionale. Ciò ha inevitabilmente creato molte aspettative, alcune delle quali sono andate deluse dal progetto Lambda, tanto che in Internet è possibile assaporare un certo malcontento leggendo nei vari blog commenti pochi lusinghieri come: si tratta di una soluzione "water down", "the poor man's Lambda expressions", e così via. Certo, forse una supporto più spinto alla programmazione funzionale sarebbe stato auspicabile, tuttavia bisogna porre attenzione a considerare attentamente i vari contesti. Scala è un linguaggio relativamente nuovo. Sebbene i lavori siano iniziati intorno al 2001 presso l'EPFL (École Polytechnique Fédérale di Losanna, Svizzera) dal professore Martin Odersky, ha veramente iniziato ad ricevere attenzioni a partire dal 2004 per poi giungere al successo conclamato nel 2011, anno in cui il team Scala ha vinto una borsa di studio per cinque anni del valore di € 2.3 M dal European Research Council (consiglio europeo di ricerca) con lo scopo di affrontare le popolari sfide poste dalla programmazione parallela. Odersky ed i suoi collaboratori hanno sfruttato l'occasione propizia per creare una startup, Typesafe, al fine di supportare commercialmente Scala con tutti i servizi annessi: training, supporto, etc. Scala quindi è nato funzionale e, fattore non trascurabile, non aveva grossissime vincoli relativi a compatibilità al ritroso, a sistemi esistenti di cui dover tener conto, librerie, application server e così via. Questo ha lasciato per lungo tempo Ordersky libero di poter cambiare radicalmente il disegno del linguaggio, alcune decisioni di base e la piattaforma stessa. All'inizio era proprio questa la critica più feroce mossa contro la Scala: l'ecosistema troppo liquido e precario. Critiche non ingiustificate visto che agli inizi non era infrequente assistere a drammatici stravolgimenti dell'ambiente e delle soluzioni di base. D'altro canto, il grande successo di Java, che reca tanti vantaggi, presenta diversi svantaggi tra cui la mancanza di queste libertà. L'evoluzione del linguaggio Java e della sua piattaforma è costretto da spazi di manovra piuttosto ristretti da molteplici, probabilmente troppi, vincoli. Basti pensare che Java continua a supportare classi come **Vector** e **Hashtable!** Quindi l'evoluzione del linguaggio

consente esclusivamente di trovare soluzioni di compromesso piuttosto che ideali. Situazione già ampliamente vissuta con J2SE 5.0 dove per esempio l'introduzione della feature dei **Generics** è stata limitata al tempo di compilazione attraverso l'esteso ricorso della strategia dell'erasure.

Con i vari limiti, tuttavia le espressioni Lambda sono comunque un notevole passo in avanti che coinvolge la quasi totalità del linguaggio di programmazione, in grado di aprire nuove frontiere, come illustrato di seguito.

Obiettivo

L'obiettivo generale del progetto Lambda è di consentire in modo idiomatico (nativo, peculiare) e semplificato l'implementazione in Java di modelli di programmazione che permettono di modellare il codice come strutture dati. Si tratta di una "filosofia", nota con i termini "Code as data", che prevede di poter manipolare opportuni blocchi di codice sorgente come una struttura di dati, e quindi come un tipo primitivo, che il linguaggio di programmazione conosce e sa manipolare. In termini accademici si parla della proprietà di **homoiconicity** (dal greco *homos*, "lo stesso", ed *eikona* che significa "rappresentazione"). Questo principio, come vedremo presto, ha delle ripercussioni molto importanti, come per esempio la possibilità di assegnare una porzione di codice ad una variabile o fornirla ad una funzione per poterla manipolare.

Il primo dibattito generatosi all'interno del progetto verteva su quale strategia adottare per la rappresentazione delle espressioni Lambda. Da un lato vi era l'appealing, ma limitata, soluzione basata sulle classi interne, dall'altro vi era una soluzione meno limitante ma tutta da inventare. Nel nel blog di Goetz ([Goetz, 2011]) è possibile trovare il seguente resoconto: "una possibilità consiste nel rappresentare le espressioni Lambda come semplici istanze di classi interne. In questo modo tutto risulterebbe piuttosto semplice ed immediato dal momento che questi elementi fanno parte del linguaggio Java fin dalle prime versioni e tutto è conosciuto. Tuttavia era evidente a tutti che si trattava di una soluzione molto limitante. Alla fine si è quindi deciso di proseguire con il concetto di funzioni. Queste sono sicuramente una direzione migliore per il futuro del linguaggio Java. La posizione di Oracle, tra l'altro, è molto chiara: Java deve evolvere, con molta attenzione, ovviamente, ma deve evolvere al fine di rimanere competitivo.

Dopo molte discussioni, l'accordo è che la migliore direzione per l'evoluzione di Java sia quella di supportare ed incoraggiare uno stile più di programmazione funzionale. Un obiettivo fondamentale del progetto Lambda è sostenere lo sviluppo e l'utilizzo di librerie funzionali. Le espressioni lambda non sono oggetti. La posizione: "lambda sono solo oggetti", sebbene sia molto comoda e

allettante, chiude la porta ad un certo numero di potenziali feature utili per l'evoluzione del linguaggio.

La decisione di percorrere la via delle funzioni ha un impatto significativo sulla piattaforma Java, come viene illustrato di seguito.

Le fondamentali feature elaborate dal progetto Lambda sono:

- **Interfacce funzionali**. Si tratta di interfacce che contengono un solo metodo astratto. L'implementazione di questo concetto, allo stato attuale richiede di definire una classe annidata anonima al fine di definire un solo metodo. Il progetto Lambda consente di definire tale metodo nella forma di un descrittore funzionale.
- **Espressioni Lambda come metodi.** Le espressioni lambda presentano molte analogie con il concetto di metodo: entrambi definiscono un elenco formale di parametri e un corpo (body) nella forma di un'espressione o di un blocco di codice tradizionale. Tuttavia, per poter veramente manipolare espressioni Lambda come metodi è necessario introdurre una serie di cambiamenti al linguaggio come illustrato nei punti seguenti.
- **Riferimenti ai metodi**. Questa feature permette di fornire il riferimento di un metodo ad un altro metodo senza necessariamente doverlo invocare immediatamente ma solo quando necessario. Questo meccanismo semplifica l'implementazione dei callback.
- **Nuove regole per la determinazione tipi di destinazione**. In molti casi, il compilatore è in grado di dedurre il tipo di destinazione attraverso il meccanismo del **type inference** (caso analogo alla feature diamond dei generics), il che significa che la stessa espressione può restituire tipi diversi in diversi contesti (polyexpression). Il tipo di ritorno viene convertito automaticamente nel tipo di destinazione corretto. Questa feature è di importante fondamentale per poter implementare espressioni lambda generiche da fornire a metodi che operare su tipi di dati specifici.

Tutte queste caratteristiche sono discusse in dettaglio nei paragrafi successivi. Tuttavia si è deciso di iniziare con un esempio pratico per comprendere da subito ed appieno l'impatto delle espressioni Lambda sul linguaggio di programmazione Java.

Tuttavia, prima di avviare questa relativamente lunga presentazione delle espressioni Lambda in Java è importante riportare le seguenti precisazioni. In primo luogo, tutte le informazioni riportate di seguito sono state attinte dalle specifiche ufficiali "JSR 335: Lambda Expressions for the Java™, Programming Language" (cfr. [Oracle, 2012]) e dai comunicati relativi allo stato del progetto ("State of Lambda"): al momento in cui viene redatto questo articolo non ci

sono fonti alternative. Inoltre, è bene ricordare che alcune aree delle espressioni Lambda sono ancora oggetto di intense discussioni e pertanto alcuni dettagli riportati di seguito potrebbero essere soggetti a cambiamenti e come Murphy insegna, sicuramente succederà.

JDK8

Al momento in cui viene redatto questo capitolo, il JDK8 è disponibile esclusivamente come preview. È scaricabile dall'URL

```
http://jdk8.java.net/lambda/
```

e non è ancora supportato da IDE come Eclipse. Quindi, per poter eseguire delle prove, è necessario tornare alle origini e compilare i sorgenti (**.java**) con il comando **javac** ed eseguire le classi (**.class**) con il comando **java**. Per la precisione, la compilazione esterna agli ambienti IDE è obbligatoria per le versioni da distribuire.

Esempio

Si consideri il compito di ordinare alfabeticamente per cognome una lista di oggetti di tipo persona (People). Con Java SE 7 è necessario dar luogo alla seguente implementazione basata su una classe annidata anonima:

```
Collections.sort(
  people,
  new Comparator<Person>() {
    public int compare(Person x, Person y) {
      return x.getLastName().compareTo(y.getLastName());
    }
  }
);
```

Listato 8. *Strategia tradizionale per implementare le interfacce funzionali*

Tale versione basata sulle classi anonime annidate è troppo prolissa, e quindi grazie all'utilizzo delle espressioni Lambda è possibile dar luogo ad una prima versione più concisa nella quale al metodo **sort** riceve come parametro, oltre la lista da ordinare, una funzione Lambda che esegue la comparazione, come riportato di seguito:

```
Collections.sort
  (people,
  (Person x, Person y) ->
     x.getLastName().compareTo(y.getLastName()));
```

Listato 9. *Prima versione del metodo Sort basato su un'espressione Lambda.*

Come si può notare, il primo risultato del progetto Lambda è di eliminare le classi annidate anonime. Tuttavia, sebbene la nuova implementazione risulti più concisa (la si può scrivere interamente su una linea!) non è più astratta ed il programmatore deve ancora definire il codice che esegue il confronto (a peggiorare la situazione interviene il caso in cui la chiave di ordinamento sia un tipo primitivo). Un serie di modifiche minori alle librerie possono fornire un grande aiuto, come ad esempio l'introduzione di un metodo di confronto (**comparing**), che prevede una funzione (**Mapper**) atta a mappare un oggetto a una chiave di ordinamento e restituire un confronto appropriato. Ciò è mostrato nel codice seguente in cui sia la definizione del metodo **comparing** sia l'interfaccia **Mapper** sono parte della libreria.

```
public <T, U extends Comparable<? super U>>
  Comparator<T> comparing(Mapper<T, ? extends U> mapper) {
    ...
}

interface Mapper<T,U> {
  public U map(T t);
}

Collections.sort(
  people,
  Collections.comparing((Person p) -> p.getLastName()));
```

Listato 10. *Versione del sort basato su librerie modificate per le espressioni Lambda.*

Questa versione può essere resa ancora più concisa demandando al compilatore il compito di determinare automaticamente (**type inference**) il tipo dei parametri dell'espressione Lambda ed importando staticamente (**import**) il metodo **comparing**. Il tutto si riduce alla seguente notazione:

```
Collections.sort(people, comparing(p -> p.getLastName()));
```

L'espressione Lambda precedente è semplicemente un meccanismo per fornire in input il metodo **getLastName**. A questo punto è possibile utilizzare i riferimenti metodo per riutilizzare il metodo esistente piuttosto che ricorrere alle espressione lambda:

```
Collections.sort(people, comparing(Person::getLastName));
```

Per finire, l'utilizzo di metodi accessori come **Collections.sort** non è desiderabile per tutta una serie di motivi: è prolisso, non può essere specializzato per ciascuna struttura di dati che implementa **List**, compromette il valore dell'interfaccia **List** dal momento che non è possibile scoprire facilmente il metodo statico di ordinamento quando si indaga la documentazione fornita dall'ambiente di sviluppo (IDE) relativa alla lista. I metodi predefiniti delle interfacce finiscono per fornire una migliore soluzione OO:

```
people.sort(comparing(Person::getLastName));
```

Questa sintassi, tra l'altro, ha un altro grande vantaggio: la notevole somiglianza alla definizione di un requisito in linguaggio pseudo-naturale: esegui l'ordinamento di una lista di persone utilizzando il cognome.

Se poi si aggiungesse un metodo predefinito **reverseOrder** al comparatore con lo scopo di produrre un comparatore che utilizza la stessa chiave ordinamento, ma in ordine inverso, si potrebbe altrettanto facilmente esprimere un ordinamento decrescente:

```
people.sort(comparing(Person::getLastName).reverseOrder());
```

Visto il grande snellimento di codice unito al significativo aumento dell'espressività, entriamo ora nei dettagli delle varie modifiche che rendono possibile supportare le espressioni Lambda.

Classi annidate anonime

In prima analisi, il paradigma OO e quello funzionale possono sembrare irreparabilmente distanti fondati su caratteristiche irriconciliabili: i linguaggi OO utilizzano oggetti che incapsulano dati e metodi, mentre quelli funzionali si basano sul concetto di funzioni. Nonostante le evidenti diversità, i rispettivi elementi base

evidentemente sono in grado di incapsulare dinamicamente il comportamento del programma e questo semplice fatto apre qualche spazio ad alcuni gradi di somiglianza e, soprattutto a possibilità di mutua influenza. Tuttavia l'opportunità aperte da questa somiglianza non è sempre immediata. Nella programmazione OO il concetto di oggetto tende ad essere relativamente pesante: si tratta di istanze di classi dichiarate in modo separato che incapsulano una serie di campi e di metodi che agiscono su di essi. Eppure, **esistono delle classi Java la cui unica responsabilità consiste nel definire un singolo metodo** (questo è il caso classico delle interfacce "callback") che per molti versi può essere visto quasi come una funzione. Si consideri l'implementazione dell'interfaccia **ActionListener** riportata di seguito:

```
public interface ActionListener {
  void actionPerformed(ActionEvent e);
}
```

Listato 11. *Interfacce che dichiarano un solo metodo.*

Una strategia frequentemente utilizzata dai programmatori Java per evitare di dover implementare un'apposita classe solo per dar luogo ad un'unica istanza utilizzata in un solo caso da una sola classe, consiste nel ricorrere al costrutto delle classi annidate anonime (anonymous inner class), come mostrato di seguito:

```
myButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    ui.dazzle(e.getModifiers());
  }
});
```

Listato 12. *Esempio di classe annidata anonima.*

Questo pattern è adottato da diverse librerie anche nel contesto della programmazione multi-threading dove è necessario assicurarsi che la parte da eseguire in parallelo sia indipendentemente dal thread che la eseguirà. Come visto nel capitolo degli aggiornamenti relativi al framework della concorrenza, il dominio del calcolo parallelo è di particolare interesse, perché attualmente i produttori di CPU sembrerebbero puntare sul miglioramento delle prestazioni ottenuto attraverso la proliferazione di processori core piuttosto che cercare di ottenere ulteriori miglioramenti dai singoli core.

Vista la crescente importanza del modello "callback" e altri idiomi basati sullo stile funzionale, è importante che Java sia in grado di supportare questi pattern nel modo più leggero possibile. La soluzione tradizionale, come visto, consiste nel ricorrere alle classi annidate anonime che però soffrono di una serie di limitazioni di cui alcune sono riportate di seguito:

- sebbene la sintassi sia più leggera rispetto alla definizione di opportune classi, resta comunque prolissa soprattutto se paragonata alle funzioni;
- esiste una certa confusione in merito al significato dei nomi e dell'utilizzo della parola chiave **this**;
- la semantica relativa al **class loading** e alla creazioni di istanze presenta una certa rigidità;
- il compilatore non riesce ad identificare variabili locali non dichiarate **final**;
- impossibilità di astrarre il controllo del flusso.

Il progetto Lambda, ovviamente, non si occupa di migliorare la sintassi delle classi annidate, tuttavia come gradito effetto collaterale, risolve diversi dei problemi succitati. In particolare:

- rimuove i punti 1 e 2 attraverso l'introduzione di nuove forme molto concise di espressione con associate regole di ambito locale;
- eludere il punto 3 attraverso la definizione della semantica delle nuove espressioni in modo più flessibile;
- migliora il punto 4 consentendo al compilatore di dedurre l'uso **final** delle variabili.

Interfacce funzionali

L'approccio basato sulle classi annidate anonime, nonostante i limiti descritti in precedenza, offre anche dei vantaggi come l'importante proprietà di integrarsi perfettamente nell'ecosistema Java. Ciò è vantaggioso per una serie di motivi: le interfacce sono già parte integrante del sistema dei tipi, hanno una rappresentazione **run-time**, portano con sé contratti informali espresse da commenti Javadoc, ecc.

L'interfaccia **ActionListener**, mostrata nel paragrafo precedente, ha un solo metodo: si tratta di un pattern comune a molte interfacce "callback". Altri esempi sono **java.lang.Runnable** e **java.util.Comparator** (per la precisione **Comparator** include anche il metodo **equals** che come riportato di seguito è considerato irrilevante in questo contesto giacché si tratta della ripetizione di un metodo definito nella classe **Object**). **Le interfacce funzionali sono esattamente tutte queste interfacce caratterizzate da un solo un metodo**. In precedenza erano chiamate tipi di **SAM: Single Abstract Method**, singolo metodo astratto.

Questa definizione non va interpretata rigidamente. Vengono considerate interfacce funzionali anche quelle che prevedono diverse versioni del metodo che dichiarano (overloading) o che ereditano da super-interfacce.

La dichiarazione di un'interfaccia funzionale non richiede speciali accorgimenti: il compilatore è in grado di identificarle come tale, in base ad un'attenta analisi della relativa struttura. Per la precisione, questo processo di identificazione non è neanche così immediato: è un po' più complesso di un semplice conteggio dei metodi presenti nella dichiarazione. Come visto, l'interfaccia potrebbe dar luogo alla dichiarazione di diverse versioni del metodo, ereditare da diversi genitori diversi metodi che però da un punto di vista logico rappresentano lo stesso, dichiarare in maniera ridondante metodi forniti automaticamente dalla classe **Object**.

Di seguito sono riportati alcuni popolari esempi di interfacce funzionali:

```
java.lang.Runnable,
java.util.concurrent.Callable,
java.security.PrivilegedAction,
java.util.Comparator,
java.io.FileFilter,
java.nio.file.PathMatcher,
java.lang.reflect.InvocationHandler,
java.beans.PropertyChangeListener, java.awt.event.ActionListener,
javax.swing.event.ChangeListener.
```

Le espressioni Lambda

La critica più severa rivolta alle classi anonime annidate è la prolissità. Le si accusa di soffrire di un problema definito " verticale": l'istanza **ActionListener** richiede cinque righe di codice sorgente per incapsulare una singola istruzione.

Le espressioni lambda sono metodi anonimi disegnati per affrontare il "problema verticale", sostituendo il meccanismo di classi interne anonime con un approccio più leggero.

Di seguito sono riportati alcuni esempi delle espressioni Lambda:

```
s -> s.length()

(int x, int y) -> x + y

() -> 42
```

```
() -> { return 42; }

(String s) -> {
  System.out.println(s);
}

() -> { System.gc(); }

(x, y, z) -> {
  if (true) {
    return x;
  } else {
    int result = y;
    for (int i = 1; i < z; i++) {
      result *= i;
    }
  return result;
  }
}
```

Listato 13. *Alcuni esempi di espressioni Lambda.*

La prima si occupa si occupa di restituire la dimensione della stringa fornita in input, la seconda esegue la somma dei due valori interi forniti in input, la terza e la quarta sono due forme equivalenti. Non richiedono alcun argomento e producono in output il numero 42, la quinta di occupa di stampare la stringa di input su console. La sesta non ha parametri, include la sola invocazione al GC (garbage collector), e restituisce un valore **void**. Per finire, l'ultima è un'implementazione piuttosto maldestra e restituisce sempre il valore della prima variabile, tuttavia è stata inclusa per mostrare che il corpo di un'espressione Lambda può essere codificato come qualsiasi metodo.

La sintassi generale prevede un elenco di argomenti, l'operatore freccia ->, e un corpo che può essere sia una singola espressione, sia un blocco di istruzioni. Da notare che la sintassi prescelta è la stessa utilizzata da C# e Scala. Le motivazioni ufficiali di questa scelta recitano che ciò è dovuto al fatto che non è stato possibile identificare un'alternativa migliore. Inoltre questa sintassi offre una serie di vantaggi, come per esempio: è già conosciuta da molti programmatori, semplifica l'interoperabilità con Scala, etc.

Nella forma semplice basata su una singola espressione, l'esecuzione è molto semplice: il corpo viene valutato e quindi ne viene restituito l'esito. Nella forma a **blocchi** la situazione non cambia di molto, il corpo viene valutato come un qualsiasi metodo: l'istruzione **return** fa sì che il controllo torni al chiamante. Parole chiave come **break** e **continue** non sono consentite al livello superiore (quello dell'espressione), mentre sono permesse in modo assolutamente naturale all'interno di cicli. Ciò significa che sono vietati salti non locali. Infine, se il corpo produce un risultato, come di consueto, ogni percorso di controllo deve restituire un valore di quel tipo o scatenare un'apposita eccezione.

La sintassi inoltre è stata ottimizzata per il caso comune in cui il corpo di un'espressione Lambda è piuttosto ridotto. Ad esempio, nella forma singola espressione viene elimina la necessità di dover includere la parola chiave **return**.

Le espressioni lambda, come logico attendersi, sono utilizzate frequentemente in invocazioni annidate: l'argomento di una chiamata è il risultato di un'altra espressione Lambda. In questi casi è possibile minimizzare l'overhead dovuto a delimitatori superflui. Tuttavia, è ancora consentito il ricorso alle parentesi per tutte quelle situazioni in cui è utile delimitare l'intera espressione, così come per qualsiasi altra espressione. Di seguito sono mostrati alcuni esempi:

```
FileFilter java =
  (File f) -> f.getName().endsWith(“.java”);

String user =
  doPrivileged(() -> System.getProperty(“user.name”));

new Thread(() -> {
  connectToService();
  sendNotification();
}).start();
```

Listato 14. *Esempi di espressioni lambda annidate.*

Tipo destinazione

La sintassi delle espressioni lambda non include il nome dell'interfaccia funzionale. Ciò implica che il tipo non è definito è quindi deve essere dedotto dal contesto. Per esempio, la seguente espressione lambda è un **ActionListener** per via del tipo di destinazione:

```
ActionListener l = (ActionEvent e) -> ui.dazzle(e.getModifiers());
```

Ciò significa che una stessa espressione Lambda può essere di **tipo diverso** in **diversi contesti** (in questo caso si parla di poly expression, multi-espressioni). Si consideri le seguenti due espressioni Lambda:

```
Callable<String> c = () -> "done";

PrivilegedAction<String> a = () -> "done";
```

Nonostante siano identiche, la prima è ovviamente di tipo **Collable**, mentre la seconda è di tipo **PriviledgedAction**. Pertanto si demanda al **compilatore la responsabilità di determinare il tipo delle espressioni lambda** (applicando la tecnica del type inference) dall'analisi del contesto dove l'espressione è inserita. Il tipo determinato viene detto target type (tipo di destinazione). Il che è complementarmente consistente con la feature del diamante introdotta con Java SE 7. Il ricorso all'inferenza di tipo implica che un'espressione Lambda può essere presente esclusivamente in contesti dove è presente un tipo di destinazione (dichiarazione di variabili, istruzioni di ritorno, assegnazioni, ecc.). Come lecito attendersi, non è fattibile/conveniente ipotizzare espressioni Lambda compatibili con ogni possibile tipo di destinazione. Pertanto, è nuovamente compito del compilatore assicurarsi che i tipi utilizzati dall'espressione Lambda siano coerenti con il tipo di destinazione dichiarato dalla firma del metodo. In particolare, un'espressione Lambda è compatibile con un tipo **T** di destinazione definito dalla firma di un metodo se tutte le seguenti quattro condizioni sono soddisfatte:

- T è un tipo di interfaccia funzionale;
- l'espressione Lambda ha lo stesso numero di parametri del metodo che dichiara il tipi di destinazione T, e tipi di tali parametri sono gli stessi;
- ogni espressione restituita dal corpo lambda è compatibile con il tipo di ritorno del metodo T;
- ogni eccezione generata dal corpo lambda è consentito dalla clausola throws del metodo T.

Dal momento che le interfacce funzionali del tipo di destinazione "sanno" quali tipi parametri formali l'espressione lambda deve ricevere, spesso non è necessario ripeterle. L'utilizzo del meccanismo dell'identificazione del tipo di destinazione consente spesso di dedurre i tipi dei parametri dell'espressioni Lambda. Come mostrato di seguito:

```
Comparator<String> c = (s1, s2) -> s1.compareToIgnoreCase(s2);
```

Il compilatore può facilmente dedurre che **s1** e **s2** sono di tipo **String** (quindi non è necessario scrivere (**String s1, String s2**)). Inoltre, in tutti quei casi in cui è presente un unico parametro il cui tipo viene dedotto (caso molto comune), le parentesi che circondano il singolo parametro possono essere omesse. Nel seguente frammento di codice sono mostrati due esempi. Il primo con l'interfaccia funzionale **FileFilter** che prevede come unico metodo **accept**(**File pathName**) e quindi evidente che il parametro può essere esclusivamente di tipo File. Nel secondo si utilizza l'interfaccia funzionale ActionListerner che definice il solo metodo **actionPerformed**(**ActionEvent e**). Quindi è facilmente desumibile che il parametro **e** sia di tipo **ActionEvent**.

```
FileFilter javaFile =
  f -> f.getName().endsWith(".java");

button.addActionListener(
  e -> ui.dazzle(e.getModifiers()));
```

Listato 15. *Espressioni Lambda con un solo parametro il cui tipo è determinato.*

Come illustrato precedentemente, le espressioni Lambda possono essere specificate esclusivamente in contesti dove è presente un tipo di destinazione. In Java questi contesti sono:

- dichiarazione di variabili;
- assegnamenti;
- istruzioni di ritorno (**return**);
- inizializzazione di array;
- argomenti di metodi e costruttori;
- corpo delle espressioni Lambda;
- espressioni condizionali (`?:`)
- istruzioni di casting.

Nei primi tre casi, il tipo di destinazione è chiaramente il tipo assegnato o di ritorno. Dall'analisi del codice seguente, si comprende che il tipo della prima espressione Lambda è un **Comparator**, mentre nel secondo caso è un **Runnable**, il cui unico metodo run invia la stringa "later" sulla console.

```
Comparator<String> c;
c = (String s1, String s2) -> s1.compareToIgnoreCase(s2);

public Runnable toDoLater() {
  return () -> {
    System.out.println("later");
  };
}
```

Listato 16. *Identificazione del tipo avviene per via dell'assegnazione.*

Per quanto concerne i contesti di inizializzazione degli array, si tratta di una particolare versione del contesto dell'assegnazione caratterizzata dal fatto che la "variabile" è un componente di matrice e il tipo è derivato dal tipo della matrice, come mostrato di seguito:

```
runAll(new Callable<String>[]{
  ()->"a", ()->"b", ()->"c"
});
```

Nel caso degli argomenti dei metodi, la situazione è più complicata: il tipo di destinazione è determinato da due caratteristiche del linguaggio di programmazione: la risoluzione dell'overloading e l'inferenza del tipo dell'argomento. Per ogni metodo potenzialmente applicabile, è compito del compilatore determinare se l'espressione Lambda è compatibile con il relativo tipo di destinazione, e di dedurre eventuali argomenti di tipo. Dopo la selezione della miglior dichiarazione di metodo, la dichiarazione fornisce il tipo di destinazione per l'espressione.

```
void invoke(Runnable r) {
  r.run();
}
<T> T invoke(Callable<T> c) {
  return c.call();
}

String s = invoke(() -> "done"); // invoke(Callable)
```

Listato 17. *Esempio di deduzione del tipo dell'argomento.*

Da notare che qualora la selezione del tipo di destinazione risulti ambigua, è sempre possibile ricorrere al casting.

Le stesse espressioni Lambda possono fornire il tipo di destinazione per i loro body, in questo caso, derivano il tipo dal tipo di destinazione esterno. Ciò fa sì che sia possibile e spesso conveniente scrivere funzioni che restituiscono altre funzioni, come mostrato di seguito. Lo stesso concetto si applica anche per le espressioni condizionali dove il tipo è fornito dal contesto.

```
Callable<Runnable> c =
  () -> () -> { System.out.println("hi"); };

Callable<Integer> c =
  flag ? (() -> 23) : (() -> 42);
```

Listato 18. *Funzioni Lambda che restituiscono altre funzioni Lambda e espressioni condizionali.*

Per finire, le istruzioni di cast forniscono un meccanismo per fornire in modo esplicito il tipo di un'espressione Lambda, qualora ciò non sia immediato dall'analisi del contesto. Le istruzioni di cast sono inoltre utili nei casi in cui sia necessario risolvere un'ambiguità dovuta alla presenza di overloading della dichiarazione di metodo con tipi di interfaccia funzionali non relazionati.

```
// Illegal: Object o = () -> { System.out.println("hi"); };
Object o = (Runnable) () -> { System.out.println("hi"); };
```

Lexical scoping (ambito lessicale)

La determinazione del significato dei nomi (e della parola chiave **this**) nelle classi annidate spesso non è immediato ed anzi è soggetto ad errori rispetto allo scenario classico di classi non annidate. Membri ereditati, tra cui i metodi derivanti dalla classe **Object**, possono accidentalmente nascondere dichiarazioni esterne, e riferimenti non qualificati a **this** si riferiscono sempre alla stessa classe interna. Le espressioni lambda sono molto più semplici: **non ereditano nomi da un super-tipo, né introducono un nuovo livello di scoping**: hanno un ambito/una portata lessicale derivata dal contesto (**lexical scoped**), ciò significa che **i nomi nel body dell'espressione sono interpretati così come se fossero**

nell'ambiente che li racchiude (con l'aggiunta dei nuovi nomi definiti dai parametri formali dell'espressioni lambda). Come logica conseguenza, la parola chiave **this** e i riferimenti ai suoi membri hanno lo stesso significato come se fossero collocati immediatamente all'esterno l'espressione lambda. Si consideri il seguente listato:

```
public class HelloWorld {
  Runnable r1 = () -> {
    System.out.println(this);
  }
  Runnable r2 = () -> {
    System.out.println(toString());
  }

  public String toString() {
    return "Hello, world!";
  }

  public static void main(String... args) {
    new HelloWorld().r1.run();
    new HelloWorld().r2.run();
  }
}
```

Listato 19. *Esempio del lexical scoping: stampa di Hello World .*

Il listato si occupa di stampare a console l'inflazionatissimo messaggio "Hello, Word!" due volte. Qualora si fosse utilizzata un'implementazione basata su classi annidate, il risultato sarebbe stato la stampa degli indirizzi di memoria **HelloWorld@...** in quanto il **this** avrebbe fatto riferimento alla classe annidata stessa.

Da notare che la parola chiave **this** all'interno delle espressioni Lambda si riferisce alla classe che le contiene **e pertanto non può essere utilizzato per riferirsi al valore calcolato dalla funzione Lambda.** In alcuni contesti, ciò genera qualche difficoltà giacché sarebbe potuto risultare utile nelle implementazioni ricorsive, fondamento della programmazione funzionale. Questo scenario è risolto consentendo, in casi ben definiti, il riferimento alla variabile a cui verrà assegnato il risultato dell'esecuzione dell'espressione come mostrato di seguito:

```
final Runnable r = () -> {
  // The following reference to 'r' is legal
  if (!allDone) {
    workQueue.add(r);
  } else {
    displayResults();
  }
};

// For contrast the following reference
// to 'objs' is illegal!!!
final Object[] objs = { "x", 23, objs };
```

Listato 20. *Esempio di riferimento alla variabile di ritorno.*

Il compilatore vieta di far riferimento alla variabile di assegnazione di un'espressione Lambda quando questa appare in contesti come un auto-assegnazione, o in un'espressione di ritorno. L'approccio corretto in questi casi consiste nel denominare l'oggetto con una dichiarazione di variabile e sostituire l'espressione originale con un riferimento variabile.

I parametri delle espressioni Lambda non possono oscurare le variabili locali nel contesto di inclusione. Ciò è coerente con l'approccio dell'ambito lessicale, e segue le regole definite per gli altri costrutti con parametri locali come per i cicli **for** e le clausole **catch**.

Acquisizione di variabili

In Java il controllo che esegue il compilatore quando esamina il codice delle classi annidate per verificare riferimenti a variabili locali delle classi che le includono (variabili acquisite) è tradizionalmente molto restrittivo: si verifica un errore ogniqualvolta la variabile acquisita non è dichiarata **final**. È possibile, e necessario, rilassare questa limitazione, sia per le espressioni Lambda, sia per classi annidate, consentendo l'acquisizione di variabili locali **effettivamente finali**. Informalmente, una variabile locale è **effettivamente finale** se il suo valore iniziale non viene mai variato, in altre parole, dichiarandola **final** non si genera alcun errore di compilazione. Riferimenti a **this**, compresi riferimenti impliciti attraverso riferimenti a campi non qualificati o invocazioni di metodi, sono, in sostanza, riferimenti a una variabile locale **final**. Corpi

delle espressioni Lambda che contengono tali riferimenti acquisiscono l'apposita istanza di **this**. Negli altri casi, nessun riferimento a questa viene mantenuto dall'oggetto. Ciò ha un risvolto benefico per la gestione della memoria: mentre le istanze della classe annidate contengono sempre un riferimento forte all'istanza che le contiene, **le espressioni Lambda** che non catturano membri dall'istanza che le racchiude **non mantengono un riferimento ad essa**. Questa caratteristica delle istanze delle classi annidate può essere fonte di **memory leak**, come per esempio il problema del **lapsed listener** (ascoltatore decaduto). Brevemente, questo problema si può originare con implementazioni non attente di modelli che traggono origine dall'**Observer Pattern**. Questi implicano un accoppiamento tra due componenti separati, l'osservatore e l'osservato, che in genere hanno cicli di vita distinti. Una delle conseguenze di registrazione di un oggetto listener è che si crea un riferimento forte tra l'oggetto osservato e degli oggetti listener, riferimento che se non gestito attentamente può impedire all'ascoltatore (e di conseguenza agli oggetti da esso riferiti) di essere rimosso dalla memoria. Questa impossibilità perdura fino a quando tutti gli ascoltatori non si deregistrano. Un'implementazione non attenta potrebbe portare al verificarsi di questa situazione. Per esempio ciò può verificarsi quando la deregistrazione è inclusa nel listener in un corpo **try...cach** e non nella parte **finally** del costrutto. Quindi qualora in tale corpo si scateni un'eccezione, ne segue che il listener non si deregistrerà mai.

Con le espressioni Lambda si intende vietare la cattura di variabili locali mutevoli. La ragione è che idiomi come quello riportato di seguito sono intrinsecamente seriali; è molto difficile scrivere corpi lambda come questo che non si prestino a **race-condition**.

```
int sum = 0;
list.forEach(e -> { sum += e.size(); });
```

A meno che non si sia disposti a far rispettare, preferibilmente in fase di compilazione, che tale funzione non possa fuoriuscire al proprio **Thread** di esecuzione, questa funzione finirebbe per causare più problemi di quanti ne possa risolvere. Un approccio migliore consiste nell'elevare il calcolo consentendo alle librerie di gestire il coordinamento tra i vari **Thread**. Nell'esempio esaminato, lo sviluppatore potrebbe utilizzare **reduce** anziché **forEach** come mostrato di seguito. La funzione **reduce** prende un valore di base, in questo caso una lista vuota, e un operatore (nell'esempio la somma) e lo esegue su tutti gli elementi (0 + list[0]+ list[1]+ ... + list[size-1])

```
int sum = list.map(e -> e.size())
       .reduce(0, (a, b) -> a+b);
```

Pertanto, piuttosto che sostenere un idioma che è fondamentalmente sequenziale ed incline a generare **race condition** sui dati (gli accumulatori che sono ovviamente mutevoli), si è preferita una libreria in grado di esprimere operazioni di accumulo in un modo parallelizzabile e meno soggetto ad errori.

Riferimenti a metodi

Le espressioni Lambda permettono di definire un metodo anonimo e trattarlo come un esempio di una interfaccia funzionale. Spesso è desiderabile fare lo stesso con metodi esistenti. **I riferimenti a metodi sono a tutti gli effetti delle espressioni che hanno lo stesso trattamento di espressioni Lambda**: hanno bisogno di un tipo di destinazione e le codifica di istanze di interfaccia funzionale, ma invece di fornire il corpo di un metodo si riferiscono a un metodo di una classe esistente o di un oggetto. Per esempio, si consideri la seguente classe **Person** i cui oggetti possono essere ordinati per nome o per età.

```
class Person {
  private final String name;
  private final int age;

  public static int compareByAge(Person a, Person b) {
    ...
  }
  public static int compareByName(Person a, Person b) {
    ...
  }
}

Person[] people = ...
Arrays.sort(people, Person::compareByAge);
```

Listato 21. *Esempio di riferimento ad un metodo.*

In questo caso l'espressione **Person::compareByAge** può essere considerata una scorciatoia per un'espressione Lambda il cui l'elenco dei parametri formali è

ripreso da **Comparator<String>** e il relativo corpo invoca **Person.compareByAge**. Poiché i tipi di parametro del metodo interfaccia funzionale hanno il ruolo di argomenti in una chiamata di metodo implicito, la firma del metodo di riferimento può modificare i parametri (come per esempio eseguendo un boxing) proprio come una chiamata di metodo.

```
interface Block<T> {
  void run(T arg);
}

// void exit(int status)
Block<Integer> b1 = System::exit;

// void sort(Object[] a)
Block<String[]> b2 = Arrays::sort;

// void main(String... args)
Block<String> b3 = MyProgram::main;

// void main(String... args)
Runnable r = MyProgram::main;
```

Listato 22. *Esempi di riferimenti a metodi statici*

Gli esempi mostrati precedentemente utilizzano metodi statici. In realtà ci sono tre diversi tipi di riferimenti ad un metodo, ognuno con una sintassi leggermente diversa:

- metodo statico;
- metodo di istanza di un determinato oggetto;
- metodo di istanza di un oggetto arbitrario di un tipo specifico.

Il riferimento ad un metodo statico, come visto, richiede di includere la classe a cui appartiene il metodo il delimitatore "**::**" e quindi il metodo.

Per quanto attiene ad un riferimento a un metodo di istanza di un oggetto specifico, il riferimento all'oggetto precede il delimitatore come mostrato di seguito:

```
class ComparisonProvider {
```

```
  public int compareByName(Person p1, Person p2) {
    ...
  }
  public int compareByAge(Person p1, Person p2) {
    ...
  }
}
...
Arrays.sort(
  people,
  comparisonProvider::compareByName);
```

Listato 23. *Riferimento ad un metodo di uno specifico oggetto*

In questo caso, l'espressione Lambda implicita catturerebbe la variabile **comparisonProvider** e il corpo invocherebbe il metodo **compareByName** utilizzando tale variabile come ricevitore. La possibilità di riferirsi ad un metodo di uno specifico oggetto fornisce un modo conveniente per convertire interfacce funzionali di diverso tipo come mostrato di seguito:

```
Callable<Path> c = ...
PrivilegedAction<Path> a = c::call;
```

Per quanto riguarda i riferimenti a metodi di istanza di un oggetto arbitrario, la sintassi prevede che il tipo a cui appartiene il metodo preceda il delimitatore ("**::**"), e il ricevitore dell'invocazione sia il primo parametro del metodo dell'interfaccia funzionale, come illustrato di seguito:

```
Arrays.sort(names, String::compareToIgnoreCase);
```

In questo caso, l'espressione Lambda implicita utilizza il suo primo parametro come ricevitore ed il secondo parametro come argomento **compareToIgnoreCase**. Se la classe del metodo di istanza è generico, i tipi dei parametri possono essere forniti prima del delimitatore ("::") o, in molti casi, possono essere dedotti dal tipo di destinazione. Si noti che la sintassi per un metodo di riferimento statico fa sì che questo possa anche essere interpretato come un riferimento ad un metodo di istanza di una classe. È compito del compilatore determinare quale sia

il caso cercando di identificare un metodo applicabile di ogni tipo (notare che il metodo di istanza ha un argomento in meno).

Per tutte le forme di riferimenti ai metodi, i tipi degli argomenti del metodo possono venir dedotti oppure, se necessario, possono essere esplicitamente inclusi dopo il delimitatore ("::").

Riferimenti al costruttore

I metodi costruttori possono essere referenziati in modo analogo ai metodi statici utilizzando l'apposita parola chiave **new**, come mostrato di seguito:

```
SocketImplFactory factory = MySocketImpl::new;
```

Qualora una classe abbia diversi costruttori, si utilizza il tipo di destinazione della firma del metodo per selezionare la migliore corrispondenza in modo analogo in cui viene risolta una chiamata di costruttore classica.

La generazione di una nuova istanza di una classe interna richiede un ulteriore parametro relativo all'oggetto esterno. Per un riferimento a costruttore, questo parametro extra può essere fornito o implicitamente racchiudendo **this** del riferimento, o può essere il primo parametro del metodo dell'interfaccia funzionale (nello stesso modo in cui il primo parametro di riferimento metodo può fungere come ricevitore di un metodo di istanza).

```
class Document {
  class Cursor {
    ...
  }

  // The enclosing instance, 'this', is implicit:
  Factory<Cursor> cursorFactory =
    Cursor::new;

  // The enclosing instance is the Mapper's parameter:
  static Mapper<Document, Cursor> DOC_TO_CURSOR =
    Cursor::new;
}
```

Listato 24. *Esempi di riferimento a costruttori.*

Metodi di default

Le espressioni Lambda e i riferimenti ai metodi apportano un notevole incremento dell'espressività del linguaggio Java. Tuttavia è necessario compiere un ulteriore passo per raggiungere l'obiettivo di un supporto nativo del modello **code-as-data.** È necessario far sì che le nuove feature siano integrate nelle varie librerie al fine di trarne pieno vantaggio. L'aggiunta di nuove funzionalità nelle esistenti librerie Java, come lecito attendersi non è assolutamente un compito immediato. In particolare, le varie interfacce, una volta rilasciate, come nel caso di Java SE 7, non possono essere più modificate. Pertanto lo scopo dei metodi predefiniti (indicati anche come **virtual extension methods**, metodi di estensione virtuali, o metodi difensivi) è quello di consentire alle interfacce di poter evolvere in modo compatibile dopo la relativa pubblicazione.

Si consideri la API delle collezioni standard la quale per supportare le espressioni Lambda dovrebbe ovviamente fornire una serie di nuove operazioni. Il metodo **removeAll**, per esempio, dovrebbe essere generalizzato al fine di consentirgli di rimuovere tutti gli elementi di una determinata collezione allorquando una data proprietà arbitraria sia soddisfatta, dove la proprietà è espressa come un esempio di un predicato di interfaccia funzionale. La domanda è dove definire questi nuovi metodi. Non si può aggiungere un metodo astratto all'interfaccia **Collection** giacché ciò genererebbe un notevole impatto in molte implementazioni esistenti. Si potrebbe ricorre ad un metodo statico nella classe di utilità **java.util.Collections**, ma ciò consisterebbe nel rilegare queste nuove operazioni in una classe in qualche modo di secondo ordine. La soluzione ottimale invece è stata considerata quella basata sui **metodi predefiniti che rappresentano un elegante disegno OO**. Si tratta di un cambiamento non banale dal momento che comporta l'aggiungere comportamento concreto ad elementi astratti per eccellenza, come le interfacce. Questo comportamento concreto è ottenibile per mezzo di nuovo tipo di metodo: un metodo di interfaccia può essere astratto, come al solito, o dichiarare un'implementazione di default.

```
interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();

  void skip(int i) default {
    for (; i > 0 && hasNext(); i--)
      next();
```

```
    }
}
```

Listato 25. *Interfaccia Iterator, metodo skip: esempio di metodo interfaccia di default.*

Dall'analisi del precedente listato ne segue che tutte le classi che implementano questa nuova definizione dell'interfaccia **Iterator** ereditano anche l'implementazione del metodo **skip**. Dal punto di vista del codice cliente, il metodo **skip** è un metodo virtuale fornito dall'interfaccia. L'invocazione del metodo **skip** su un'istanza di una classe che implementa, direttamente o indirettamente, **Iterator**, genera l'invocazione dell'invocazione predefinita. Ciò genera il beneficio che le "sottoclassi" di **iterator**, a meno di esigenze particolari, non devono preoccuparsi di ridefinire l'implementazione di questo metodo. È appena il caso di sottolineare che le varie classi sono liberissime di ignorare l'implementazione standard di un metodo per implementarne una versione diversa, per esempio nel caso di **skip** si potrebbe avere la necessità di gestire direttamente un cursore privato, o per incorporare una sofisticata politica **thread-safe**. In modo analogo, quando un'interfaccia ne estende un'altra, può aggiungere, modificare o rimuovere le implementazioni predefinite dei metodi della interfaccia genitore. Al fine di consentire ad una interfaccia di rimuovere un'implementazione predefinita, è necessario introdurre la nuova parola chiave: **none**.

Ereditarietà dei metodi di default

Metodi predefiniti vengono ereditati così come avviene per gli altri metodi e nella maggior parte dei casi, il comportamento è proprio quello che ci si aspetta. Tuttavia, in alcune circostanze particolari, è necessario qualche chiarimento.

In primo luogo, quando un'interfaccia ridichiara un metodo di uno dei suoi super-tipi (ripete la firma del metodo) senza menzionare la parola chiave **default**, allora anche l'implementazione di default, se presente, viene ereditata dalla interfaccia che ha incluso l'override. Per capire la motivazione di questa scelta è necessario considerare una pratica di documentazione utilizzata di frequente che consiste nel ridichiarare dei metodi. Si vuole, pertanto, evitare che la semplice ripetizione di un metodo che già implicitamente è un membro dell'interfaccia generi effetti collaterali inattesi.

In secondo luogo, quando un tipo genitore di una classe o di un'interfaccia definisce indirettamente diversi metodi con la stessa firma, le regole di ereditarietà tentano di risolvere il conflitto. I seguenti due principi fondamentali che si applicano sono quelle che seguono.

Le **dichiarazioni** di metodi nelle classi sono preferiti ai metodi di default predefiniti nelle interfacce. Questo è vero indipendentemente dal fatto che il metodo della classe sia concreto o astratto. Alla parola chiave default deve essere associato sempre un significato di ripiego qualora la gerarchia delle classi non ridefinisca il metodo.

Metodi **già sovrascritti** da altri candidati sono ignorati. Questa circostanza può verificarsi quando super-tipi condividono un medesimo antenato. Si consideri il caso in cui le due interfacce per esempio Collection e List (List eredita da Collection) forniscano diverse implementazioni di default per i metodi removeAll. In questo scenario, la clausola implements riportata nella dichiarazione di seguito fa sì che il metodo di default definito nell'interfaccia List abbia la precedenza, e quindi venga ereditata da Queue, sulla corrispondente dichiarazione definita dall'interfaccia Collection:

```
class LinkedList<E> implements List<E>, Queue<E>
```

Nel caso in cui due metodi di **default**, definiti in modo indipendente generino un conflitto, o nella situazione analoga in cui il conflitto è generato da un metodo di **default** e un corrispondente **none** (un'attuazione del famoso atavico problema del diamante), il programmatore deve includere esplicitamente l'**override** dei metodi dei super-tipi che generano il confitto (da notare che questo, in qualche misura, significa accettare elementi di ereditarietà multipla in Java). Nella maggior parte dei casi, ciò dovrebbe risolversi nella selezione del default preferito. Una sintassi migliore consiste nel ricorrere alla parola chiave **super** per selezionare l'invocazione di implementazione predefinita di una particolare super interfaccia (seguendo la strategia utilizzata da C++). In particolare, è possibile prevedere la definizione dell'implementazione di default da utilizzare inserendo il nome della classe, poi super ed infine il metodo in questione, come mostrato nel seguente listato.

```
interface Robot implements Artist, Gun {
  void draw() default {
    Artist.super.draw();
  }
}
```

Listato 26. *Risoluzione di un conflitto nell'eredità di metodi di default.*

Sebbene questa soluzione presenti diversi vantaggi, in qualche modo rappresenta una **deviazione della decisione base del linguaggio Java basata sulla singola ereditarietà**. Ovviamente è necessario far sì che il linguaggio evolva e non c'è nulla di male nel rivedere alcune scelte iniziali, così come è avvenuto già in diversi ambiti (multi-threading, java.io. etc.). Tuttavia, in questi casi, probabilmente potrebbe valere la pena rivedere l'intera policy e non solo i singoli casi.

java.util.functions

Le espressioni Lambda in Java sono convertite in istanze di interfacce con un solo metodo (interfacce funzionali). Per supportare questo concetto è necessario disporre di una serie di interfacce funzionali di base. Queste sono state incluse nel nuovo package: java.util.functions e sono i blocchi base utilizzati dalle varie librerie. Le più interessanti sono riportate di seguito.

- **Predicate** (Predicate<T>) : verifica che la specifica proprietà sia verificata dall'oggetto fornito in input;
- **Block** (Block<T>) : rappresenta un'azione da eseguire sull'oggetto specificato come parametro;
- **Mapper** (Mapper<T, U>) : si occupa di trasformare elementi appartementi al dominio T in elementi appartenenti al dominio U;
- **UnaryOperator** (UnaryOperator<T>) : permette di rappresentare operatori unari in cui sia l'elemento di input (dominio), sia quello di output (codomio) appartengono allo stesso insieme;
- **BinaryOperator** (BinaryOperator<T, T>): come al punto precedente, con la differenza che l'operatore prende due parametri in input dello stesso dominio anziché uno.

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

  // Predicate. (Integer) -> (Boolean)
  System.out.println("--------------- Predicate");
  Predicate<Integer> isEven = n -> n % 2 == 0;
  for (int i=1; i<7; i++) {
    System.out.println("Is "+i+" even?"+isEven.test(i) );
  }
  System.out.println("-------------------------\n");

  // Unary Operator (Double) -> (Double)
  System.out.println("--------------- Unary Operator");
```

```
UnaryOperator<Double> celsius = n -> (n-32) * 5 / 9;
UnaryOperator<Double>  fahrenheit = n -> n * 9 / 5  + 32;
System.out.println("0C="+fahrenheit.operate(0d)+"F" );
System.out.println("100C="+fahrenheit.operate(100d)+"F");
System.out.println("32F="+celsius.operate(32d)+"C" );
System.out.println("212F="+celsius.operate(212d)+"C" );
System.out.println("------------------------- \n");


// Binary Operator (Integer, Integer) -> Integer
System.out.println("--------------- Binary Operator");
BinaryOperator<Integer> power = (b, e) -> {
  int result = b;
  if (e == 0) {
    result = 1;
  } else {
  for (int i=1; i<e; i++) {
     result *= b;
    }
  }
  return result;
};

for (int i = 0; i < 8; i++) {
  System.out.println(
    "power 2^"+i+"="+power.operate(2, i));
}
System.out.println("-------------------------\n");


// comparator. Integer is the return type
System.out.println("--------------- Compare");
Comparator<Integer> compare = (first, second) -> {
  return first > second ? 1: (first < second ? -1 : 0);
};
System.out.println(
  "Compare(10,10) ="+compare.compare(10, 10) );
System.out.println(
```

```
    "Compare(5,9) ="+compare.compare(5, 9) );
  System.out.println(
    "Compare(9,5) ="+compare.compare(9, 5) );
  System.out.println("------------------------\n");

  // mapper (String) -> (Integer)
  System.out.println("--------------- Mapper");
  Mapper<String, Integer> aMapper = s -> s.length();
  System.out.println(
    "'Lambda'. Size="+aMapper.map("Lambda") );
  System.out.println(
    "'JavaSE8'. Size="+aMapper.map("JavaSE8") );
  System.out.println("------------------------\n");
}
```

Listato 27. *Semplice esempio di utilizzo delle interfacce funzionali di base.*

Dall'analisi del listato è possibile notare l'operazione di deduzione del tipo eseguita dal compilatore. Per esempio, il **Predicate isEven** non è stato scritto nella forma:

```
Predicate<Integer> isEven = (Integer n) -> n % 2 == 0;
```

Così come l'**UnaryOperator celsius** non è stato scritto nella forma:

```
UnaryOperator<Double> celsius = (Double n) -> (n-32) * 5 / 9;
```

L'operatore power non è stato scritto nella forma:

```
BinaryOperator<Integer> power = (Integer b, Integer e) -> {
```

L'esecuzione del precedente listato, genera l'output riportato di seguito.

```
--------------- Predicate
Is 1 even?false
Is 2 even?true
Is 3 even?false
```

```
Is 4 even?true
Is 5 even?false
Is 6 even?true
-------------------------
--------------- Unary Operator
0C=32.0F
100C=212.0F
32F=0.0C
212F=100.0C
-------------------------
--------------- Binary Operator
power 2^0=1
power 2^1=2
power 2^2=4
power 2^3=8
power 2^4=16
power 2^5=32
power 2^6=64
power 2^7=128
-------------------------

--------------- Compare
Compare(10,10) =0
Compare(5,9) =-1
Compare(9,5) =1
-------------------------
--------------- Mapper
‘Lambda’. Size=6
‘JavaSE8’. Size=7
-------------------------
```

Ricorsione

Dall’analisi del precedente listato si può notare che l’operatore di elevamento a potenza è stato implementato in forma iterativa. Trattandosi di programmazione funzionale, sarebbe stato più elegante riportare un’implementazione ricorsiva. **Nelle espressioni Lambda Java la ricorsione non sembra né immediata, né elegante** poiché la si può utilizzare soltanto se la chiamata ricorsiva fa riferimento ad un nome definito nel contesto che racchiude la definizione dell’espressione

Lambda. Più precisamente (nel momento in cui viene scritto questo capitolo) le definizioni ricorsive possono essere incluse solo in un contesto di assegnamento di variabile e, per via della regola dell'assegnamento prima dell'uso, solo in variabili di istanze o assegnamento di variabili definite statiche, come illustrato di seguito.

```
public class Recoursion {

  static FactInt factorial;

  public static void main(String[] args) {

    factorial = i -> i == 0 ? 1 : i * factorial.invoke(i - 1);

    for (int ind=0; ind < 10; ind++) {
      System.out.println(
        "Factorial "+ind+"="+factorial.invoke(ind));
    }
  }
}

public interface FactInt {
   int invoke(int i);
}
```

Listato 28. *Implementazione ricorsiva dell'operatore fattoriale.*

L'esecuzione del precedente listato genera il seguente output:

```
Factorial 0=1
Factorial 1=1
Factorial 2=2
Factorial 3=6
Factorial 4=24
Factorial 5=120
Factorial 6=720
Factorial 7=5040
```

```
Factorial 8=40320
Factorial 9=362880
```

Interazioni interne ed esterne

Il framework delle collezioni si basa sul concetto di iterazione esterna: **le collezioni forniscono metodi** che possono essere utilizzati per **scorrere i propri elementi** (per esempio le interfacce **Set**, **List** e **Queue** ereditano da **Collection**, la quale a sua volta eredita da **Iterable<E>**). Il codice delle classi client utilizza questi metodi per scorrere in sequenza gli elementi della collezione. Per esempio, il codice seguente mostra un ciclo atto ad impostare ad un valore predefinito lo stato di una serie di elementi:

```
for(Block currentBlock : names){
  currentBlock.setAvailable();
}
```

Listato 29. *Semplice iterazione esterna.*

Il listato precedente mostra un semplicissimo esempio di iterazione esterna, il ciclo **for-each** invoca i metodi dell'iteratore al fine di scorrere attraverso tutti gli elementi: uno per uno. L'iterazione esterna è molto semplice, tuttavia non è immune da una serie di problemi come per esempio:

1. In Java è intrinsecamente seriale dal momento che Java non fornisce alcuno strumento per consentirne una elaborazione parallela.
2. Non prevede la possibilità di disporre di meccanismi atti a gestire il flusso di controllo, che potrebbero fornire servizi di riordino dei dati, esecuzione parallela, corto circuito dei cicli, esecuzioni lazy atte a migliorare le performance, etc. Ovviamente, esistono dei casi in cui la specifica di stretto rispetto dello scorrimento sequenziale e nel dato ordine è fondamentale, tuttavia in molti altri casi ciò è un impedimento alla performance e alla chiarezza del codice.

L'alternativa alla iterazione esterna è l'opposto, ossia l'iterazione interna, dove invece di controllare l'iterazione, il codice cliente delega il ciclo alla libreria standard fornendo il codice da eseguire durante l'iterazione, come mostrato nel seguente frammento di codice:

```
devices.forEach(currentDevice -> { currentDevice.setAvailable();
});
```

L'approccio dell'iterazione interna **sposta il flusso di controllo di gestione dal codice cliente a quello della libreria**. Ciò produce una serie di benefici che in buona sostanza risolvono i limiti delle iterazioni esterne indicati al punto 2. Quindi permette di alternare il ciclo di scorrimento, permette di introdurre in modo elegante la strategia del corto circuito, dell'elaborazioni lazy, etc. Lo stile dell'iterazione interna inoltre si presta ad uno stile di programmazione in cui le operazioni possono essere concatenate. Per esempio, il seguente codice mostra il caso in cui si voglia richiedere ai device in stato di **stand-by**, di transitare allo stato di disponibile.

```
devices
  .filter(currentDevice -> currentDevice.isStandBy())
  .forEach(currentDevice -> {
     currentDevice.setAvailable();
  });
```

Listato 30. *Esempio di utilizzo dell'operazione filter.*

L'operazione di filtro produce un flusso costituito dai valori che soddisfano la condizione specificata per mezzo dell'espressione Lambda fornita come parametro, tale flusso è poi fornito all'operatore **forEach**. Qualora si volesse copiare i device che si trovano in uno stato di non disponibilità, sarebbe sufficiente utilizzare in modo del tutto naturale l'operatore **into**, come mostrato di seguito:

```
List<Device> standByDevices =
  devices
    .filter(currentDevice -> !currentDevice.isAvailable())
    .into(new ArrayList<>());
```

Listato 31. *Esempio di utilizzo dell'operazione into.*

Come si può notare il ricorso all'iterazione interna permette di dar luogo a codice più vicino al linguaggio naturale e quindi più leggibile, concatenare le varie operazioni, fornire dei plug-in per il parallelismo (come mostrato di seguito), etc.

Pigri e avidi

I vari cicli tipici della programmazione possono essere eseguiti secondo due strategie: pigra (**lazy**), o avida (**eager**). Come lecito attendersi, non esiste una

soluzione che soddisfi tutti i casi. In alcune situazioni, dove è necessario scorrere subito tutti i valori per produrre un risultato, un approccio **eager** offre migliori prestazioni. Mentre in altri casi, quando è desiderabile gestire l'utilizzo di risorse (incluso il tempo) in maniera più graduale evitando la creazione di oggetti temporanei, l'approccio opposto, quello pigro, risulta la scelta migliore. Si consideri per esempio un'operazione di filtraggio che dovrebbe venir eseguita solo quando si avvia l'iterazione sugli elementi risultato del filtro stesso. Operazioni di stream che producono nuovi flussi dati, come il **filter** e **map**, si prestano naturalmente ad essere implementati con un approccio pigro, che tende a portare miglioramenti significativi delle prestazioni. D'altra parte, operazioni come l'accumulo, o quelli che producono effetti collaterali, quali il dumping dei risultati in una collezione o che eseguono qualche operazione su ogni elemento (come ad esempio l'impostazione dei valori, l'output in un log), tendono naturalmente ad essere implementati in modo avido.

L'approccio deciso per le espressioni Lambda consiste nell'estendere l'interfaccia **java.lang.Iterable<T>**. Da notare che fino a Java SE 7 questa interfaccia definiva un solo metodo: **iterator**() il cui fine è ritornare un **Iterator** per scorrere gli elementi di tipo **T**. La nuova versione prevede l'inclusione sia di metodi che implementano una strategia **lazy** sia metodi basati sulla strategia **eager** come mostrato dal listato 32.

```
public interface Iterable<T> {

  // Abstract methods
  Iterator<T> iterator();

  // Lazy operations
  Iterable<T>
  filter(Predicate<? super T> predicate) default ...

  <U> Iterable<U>
  map(Mapper<? super T, ? extends U> mapper) default ...

  <U> Iterable<U>
  flatMap(Mapper<? super T, ? extends Iterable<U>> mapper) default
...

  Iterable<T>
```

```
  cumulate(BinaryOperator<T> op) default ...

  Iterable<T>
  sorted(Comparator<? super T> comparator) default ...

  <U extends Comparable<? super U>> Iterable<T>
  sortedBy(Mapper<? super T, U> extractor) default ...

  Iterable<T>
  uniqueElements() default ...

  <U> Iterable<U>
  pipeline(Mapper<Iterable<T>, ? extends Iterable<U>> mapper)
default ...

  <U> BiStream<T, U>
  mapped(Mapper<? super T, ? extends U> mapper) default ...

  <U> BiStream<U, Iterable<T>>
  groupBy(Mapper<? super T, ? extends U> mapper) default ...

  <U> BiStream<U, Iterable<T>>
  groupByMulti(Mapper<? super T, ? extends Iterable<U>> mapper)
default ...

  // Eager operations

  boolean isEmpty() default ...;
  long count() default ...

  T getFirst() default ...
  T getOnly() default ...
  T getAny() default ...

  void
  forEach(Block<? super T> block) default ...

  T
```

```
    reduce(T base, BinaryOperator<T> reducer) default ...

    <A extends Fillable<? super T>> A
    into(A target) default ...

    boolean
    anyMatch(Predicate<? super T> filter) default ...
    boolean
    noneMatch(Predicate<? super T> filter) default ...
    boolean
    allMatch(Predicate<? super T> filter) default ...

    <U extends Comparable<? super U>> T
    maxBy(Mapper<? super T, U> extractor) default ...

    <U extends Comparable<? super U>> T
    minBy(Mapper<? super T, U> extractor) default ...
}
```

Listato 32. *Prototipo dell'interfaccia Iterable.*

Come mostrato dal listato, l'approccio selezionato permette di trattare operazioni naturalmente pigre in modo da far restituire un flusso (ad esempio **Iterable**) piuttosto che una nuova collezione. Quest'ultimo approccio è stato valutato e avrebbe potuto portare vantaggi in situazioni di comportamento avido, risultando invece problematico negli altri casi. In particolare, la generazione di collezioni create ad esclusivo uso e consumo della catena di invocazioni e quindi, verosimilmente, deferenziata già dall'invocazione successiva della catena, avrebbe finito per dar luogo ad un pessimo utilizzo delle risorse tempo e memoria. La strategia basata sull'iteratore funziona perfettamente in tutti questi casi nei quali si opera su flussi, come per esempio **filter** che parte da una sorgente dati (che a sua volta potrebbe essere un altro flusso) e produce un nuovo flusso con i valori che soddisfano la condizione specificata come parametro. In alcuni casi specifici in cui un funzionamento eager sarebbe stato ideale, questa strategia funziona ancora bene, ma forse non è ottimale, anche per questo, questa scelta è stata annotata come "da rivedere". Il ricorso a **Iterable** è stata l'astrazione selezionata per gli stream di dati nelle invocazioni concatenate. Tuttavia, questo approccio ha il vantaggio che, se usato in una concatenazione **sorgente dati-lazy-lazy-eager**,

la parte lazy è per lo più invisibile, la pipeline risulta "ben sigillata" da entrambi i lati, e produce sia una buona usabilità, sia buone prestazioni senza aumentare drammaticamente il livello di astrazione concettuale delle librerie.

Al fine di comprendere la potenza della nuova interfaccia, si consideri il task di produrre in output un elenco ordinato alfabeticamente con i nomi degli album musicali che includono almeno una traccia con un punteggio almeno uguale a quattro. Questo breve esercizio, in Java SE 7, avrebbe richiesto di produrre un codice simile a quanto riportato nel listato 33. Da notare che il codice include una classe annidata anonima per risolvere il compito della comparazione ed un corto circuito poco elegante (break) e decisamente non strutturato.

```
List<Album> favs = new ArrayList<>();
for (Album a : albums) {
  boolean hasFavorite = false;
  for (Track t : a.tracks) {
    if (t.rating >= 4) {
      hasFavorite = true;
      break;
    }
  }
  if (hasFavorite) {
    favs.add(a);
  }
}
Collections.sort(
  favs,
  new Comparator<Album>() {
    public int compare(Album a1, Album a2) {
      return a1.name.compareTo(a2.name);
    }});
```

Listato 33. *Codice tradizionale per individuare la lista degli albumi preferiti.*

Lo stesso metodo implementato con le espressioni Lambda diviene decisamente conciso e più elegante come mostrato nel listato 34.

```
List<Album> sortedFavs =
  albums.filter(
```

```
        a -> a.tracks.anyMatch(t -> (t.rating >= 4)))
      .sortedBy(a -> a.name)
      .into(new ArrayList<>());
```

Listato 34. *Stesso compito del listato precedente implementato con le espressioni Lambda.*

Parallelismo

Come riportato nei precedenti paragrafi, il ricorso alle iterazioni interne consente di eseguire le varie operazioni in parallelo. Tuttavia la strategia scelta è stata di **non dar luogo ad un comportamento parallelo trasparente**, ossia di non demandare la decisione al compilatore e/o alla JVM, ma di lasciare tale decisione al programmatore. Quindi **gli sviluppatori hanno il compito di selezionare il comportamento parallelo in modo esplicito ma non invadente.** A tal fine è stata introdotto il metodo **parallel**(), proprio per permettere agli sviluppatori di indicare le operazioni da eseguire in parallelo come mostrato dal listato successivo.

```
int sum =
  devices .parallel()
       .filter(currDevices -> currDevices.isAvailable())
       .map(currDevices -> currDevices.getWorkLoad())
       .sum();
```

Listato 35. *Esegue la somma del work load dei dispositivi disponibili.*

Come visto nel Capitolo 3 "Fork et impera", Java SE 7 ha introdotto il framework Fork/Join che rappresenta un ottimo meccanismo per risolvere problemi di natura **divide and conquer**. Si tratta del **candidato ideale per implementare comportamenti paralleli per le espressioni Lambda**. Tuttavia, la sua applicazione in questo contesto non è assolutamente immediata. In particolare, senza lavoro aggiuntivo, finirebbe per minare uno degli obiettivi base che consiste nel ridurre al minimo il divario tra la sintassi delle implementazioni Lambda seriali e le corrispondenti parallele: tale differenza è stata pensata per essere ridotta alla presenza o meno del metodo **parallel**. Il framework Fork/Join prevede una sintassi significativamente diversa dal codice richiesto per implementazioni seriali. Questa lacuna è colmabile attraverso la definizione di due interfacce: **Splittable** e **ParallelIterable**. La prima si occupa di definire dei metodi che permettono

il mapping trasparente al framework Fork/Join. In particolare, permette di suddividere il problema in sottoproblemi da eseguire in parallelo per poi combinare i risultati.

```
public interface Splittable<T, S extends Splittable<T, S>> {
  /* * Return an {@link Iterator}
    * for the elements of this split.
    * In general, this method is only called
    * at the leaves of a decomposition tree,
    * though it can be called at any level.
    */
  Iterator<T> iterator();

  /* * Decompose this split into two splits,
    * and return the left split.
    * If further splitting is impossible,
    * {@code left} may return a {@code Splittable}
    * representing the entire split, or an empty split.
    */
  S left();

  /* * Decompose this split into two splits,
    * and return the right split.
    * If further splitting is impossible,
    * {@code left} may return a {@code Splittable}
    * representing the entire split, or an empty split.
    */
     S right();

  /* *
    * Produce an {@link Iterable} representing
    * the contents of this {@code Splittable}.
    * In general, this method is only called at
    * the top of a decomposition tree,
    * indicating that operations that produced
    * the {@code Spliterable} can happen in parallel,
    * but the results are assembled for sequential traversal.
    * This is designed to support patterns like:
```

```
     * collection
     *   .filter(t -> t.matches(k))
     *   .map(t -> t.getLabel())
     *   .sorted()
     *   .sequential()
     *   .forEach(e -> println(e));
     *
     * where the filter/map/sort operations can occur in parallel,
     * and then the results can be traversed sequentially
     * in a predicatable order.
     */
   Iterable<T> sequential();
}
```

Listato 36. *Interfaccia Splittable.*

L'interfaccia **ParallelIterable** è in sostanza l'equivalente di Iterable che include comportamento parallelo e quindi include il comportamento della precedente interfaccia.

Conclusioni

In questo capitolo abbiamo presentato i principali progetti che dovrebbero confluire in Java SE 8: le molteplici proposte del progetto **Coin** ("moneta") escluse da Java SE 7, la realizzazione del progetto di modularizzazione di Java **Jigsaw** ("puzzle"), l'introduzione delle **Lambda Expression**. Il condizionale è quanto mai d'obbligo giacché già si parla di modifiche sul progetto iniziale e possibili posticipazioni a Java SE 9 (vedi il progetto **Jigsaw**).

Sebbene al momento in cui viene redatto questo capitolo Java SE 8 è ancora in sviluppo, e quindi alcune parti potrebbero subire dei cambiamenti (e ci potete contare che succederà), è lecito attendersi che le parti core non cambino significativamente e comunque l'analisi dello stato attuale resta un ottimo studio sull'evoluzione Java. Per quanto attiene il progetto Jigsaw, sono state prese in considerazione ben sessanta proposte (oltre quelle già consegnate con Java SE 7) di queste, ovviamente, sarà possibile includerne solo un limitato sottoinsieme. Tra le più interessanti vale la pena menzionare: Elvis and Other Null-Safe Operators che permette di migliorare notevolmente la qualità del codice semplificando i ricorrenti controlli delle referenze nulle, static methods in interface, che consiste

nel rendere possibile incapsulare metodi statici nelle interfacce al fine di realizzare codice più OO riducendo la necessità di ricorrere a classi ausiliari, large array ossia il supporto per gli array di grandi dimensioni, sintassi più concisa per l'inizializzazione delle collezioni e un supporto più conciso per eseguire il test di uguaglianza e il supporto per diversi tipi di annotazione.

Lo scopo del progetto Jigsaw è disegnare e realizzare un sistema modulare standard per la piattaforma Java SE da applicare sia alla stessa piattaforma, sia al JDK. Si tratta di realizzare un sistema che sia una sorta di mix tra Maven e OSGi. Da una parte Maven da solo non è sufficiente in quanto si tratta di un tool per la gestione e comprensione dei progetti e quindi è uno strumento che per sua natura, si limita agli aspetti dei progetti relativi a tempo di costruzione. Dall'altro lato, anche OSGi, da solo non è sufficiente in quanto si tratta di un sistema modulare e di una piattaforma di servizi per Java che implementa un modello a componenti completo e dinamico limitandosi agli aspetti di modularizzazione a tempo di esecuzione. Almeno da un punto di vista teorico, Jigsaw può essere visto come un'opportuna combinazione tra Maven e OSGi, combinazione già realtà in diverse organizzazioni. Il limite di questa strategia è che si tratta di una soluzione creata a partire dalla piattaforma Java, mentre una vera innovazione dovrebbe richiedere cambiamenti più radicali a partire dalla stessa piattaforma. Jigsaw è in forte ritardo rispetto alla pianificazione iniziale, il che è anche comprensibile visto e considerato l'enorme impatto sull'intero ecosistema Java. Tanto che Mark Reinhold, Chief Architect del Java Platform Group presso Oracle, nel suo blog ha già avanzato la proposta per il suo discoping.

Sebbene i precedenti progetti esercitino un certo appeal sulla comunità Java, la grande attesa è tutta rivolta al progetto Lambda il cui obiettivo è quello di introdurre finalmente la programmazione funzionale in Java, seguendo (meglio rincorrendo) un po' le orme di Scala e C#. L'introduzione di tale supporto è un esercizio delicato e complicato allo stesso tempo. Molti sostenitori Java sembrerebbero essere rimasti delusi dall'approccio per così dire "conservatore" adottato dagli architetti Java, tanto che in Internet è possibile assaporare un certo malcontento. D'altro canto è necessario considerare che il grande successo di Java reca molteplici vantaggi, ma, al tempo stesso, presenta diversi svantaggi tra cui una certa limitazione che costringe l'evoluzione di Java e della sua piattaforma in spazi di manovra piuttosto angusti. Esistono molteplici, probabilmente troppi, vincoli (tra cui il cavallo di battaglia dell'eccessiva compatibilità con il passato). Quindi l'evoluzione del linguaggio spesso consente di trovare esclusivamente soluzioni di compromesso piuttosto che ideali. Con i suoi limiti, tuttavia le espressioni Lambda restano un notevole passo in avanti che coinvolge la quasi totalità

del linguaggio di programmazione apportando una sintassi più elegante e potente e, da non sottovalutare, un avanzato supporto a diversi aspetti della programmazione parallela. Per molti addetti ai lavori, la programmazione funzionale è il futuro di Java, futuro che tuttavia richiede un'attenta evoluzione.

I principali cambiamenti necessari per supportare le espressioni Lambda sono quelli riportati di seguito.

Interfacce funzionali. Interfacce che contengono un solo metodo astratto che, come tali, dovrebbero decretare la morte delle classi anonime annidate utilizzate come implementazione del pattern del call-back.

Espressioni Lambda come metodi. Questa è una vera rivoluzione che permette di trattare funzioni (ossia elementi simili a metodi) come se fossero tipi primitivi del linguaggi: **code as data**.

Riferimenti ai metodi. Questa feature permette di fornire il riferimento di un metodo ad un altro metodo senza necessariamente doverlo invocare immediatamente ma solo quando necessario.

Nuove regole per la determinazione tipi di destinazione. In molti casi, il compilatore è in grado di dedurre il tipo di destinazione attraverso il meccanismo del **type inference**, il che significa che la stessa espressione può restituire tipi diversi in diversi contesti.

La domanda che si pongono in molti è se l'inclusione delle espressioni Lambda sia in grado effettivamente di colmare il gap con Scala. Arginarne l'attuale espansione dovuta anche all'estrema semplificazione della programmazione parallela. Attualmente è possibile venire coinvolti in dibattiti interni ad aziende intente a migrare progetti Java in Scala e pure il viceversa: aziende che si sono gettate a testa bassa su Scala per poi tornare sui propri passi e progettare una migrazione a ritroso.

Scala è indubbiamente un linguaggio moderno, che si è evoluto drasticamente ignorando la compatibilità con il passato che gli ha permesso importanti mutamenti in corso d'opera e soprattutto supporta tutte le feature (e anche di più) che in Java verranno introdotte solo nel 2013-14. Inoltre è importante tener presente che sebbene abbia ottenuto diverse sovvenzioni, non ha dietro grandi aziende come Oracle e Microsoft. Detto ciò, in diversi contesti, come per esempio l'implementazione di particolari calcoli del rischio, può avere molto senso dar luogo ad implementazioni direttamente su Scala, mantenendo però una forte base Java. Come al solito, scelte di questo tipo sono complesse e richiedono considerazioni di diversa natura non strettamente di carattere tecnico. È necessario considerare la qualità e dinamicità del personale a disposizione, la facilità/difficoltà nel reperire gli skill necessari, le learning curve, la strategicità dei progetti, la comunità

esistente, la presenza di una governance ben collaudata, e così via. Un elemento da non trascurare è che è possibile integrare codice scritto in Scala in applicazioni Java. Quindi è ancora attuale e consigliabile mantenere una forte presenza in Java, con alcune deviazioni in contesti ben definiti e controllati.

Per finire, il progetto Lambda con il suo supporto funzionale dovrebbe portare benefici anche a Scala che nella maggioranza dei casi gira su JVM!

Appendice A
Multi-Threading in Java

Prefazione

Questa appendice presenta una rapida disamina dell'evoluzione della strategia del MT (multi-threading) in Java. L'obiettivo non è tanto quello di presentare i vari aspetti, le diverse problematiche, i numerosi tranelli e le possibili soluzioni inerenti al MT (per questo si rimanda all'"Appendice D. Multi-threading" del libro "Java Best Practices", cfr. [Vetti Tagliati, 2008]) ma di illustrare brevemente come tale filosofia si sia evoluta nel tempo, attraverso le varie release. Detto ciò, vengono anche presentati brevemente alcuni meccanismi del package **java.util.concurrent** giacché la loro conoscenza è propedeutica alla piena comprensione delle nuove feature descritte nel Capitolo 3 "Fork et Impera".

Tutti coloro che hanno seguito l'evoluzione del linguaggio Java fin dalla sua genesi, ricordano la straordinaria (ma, con il senno di poi, poco ottimale) enfasi conferita alla programmazione MT. Ciò si evince da una serie di elementi nativi del linguaggio, quali: la presenza di specifiche classi e interfacce disegnate appositamente per il MT e la disponibilità di determinate parole chiave e metodi dedicati alla programmazione MT. Per esempio, affinché i metodi (delle istanze) di una classe possano essere eseguiti all'interno di un apposito T (Thread) dedicato, è sufficiente far sì che la classe implementi l'interfaccia Runnable (e quindi definisca l'implementazione del metodo run()), oppure, in alternativa, è possibile estendere direttamente la classe java.lang.Thread. Ciò rende possibile definire una serie di T la cui esecuzione compete con quella del metodo principale del programma.

All'inizio la comunità Java (e non solo) era pressoché unanime nel considerare questa scelta progettuale come uno dei grandi punti di forza di Java. Ben presto tuttavia si capì che le cose forse non erano esattamente così e che probabilmente si trattò di un errore. Per comprendere da un punto di vista pratico questa situazione è possibile paragonare la strategia iniziale all'acquisto di uno mini-stereo, ossia ad uno quei stereo che include i vari componenti: lettore di DVD/CD, MP3, amplificatore, radio, etc. Sebbene ciò all'inizio possa sembrare un grande vantaggio: tutto è pronto a funzionare, basta premere un pulsante,

tale l'entusiasmo iniziale è destinato a terminare presto, quando si cominciano a comprendere le limitazioni: non è possibile sostituire le varie componenti con implementazione più moderne e scalabili.

Breve storia

Fin dai tempi di Oak (versione primordiale di Java) la decisione progettuale fu di dotare il linguaggio di primitive, interfacce e classi per supportare nativamente il MT. Ciò permette di poter definire semplicemente una serie di T che si contendono i tempi di esecuzione sulla JVM (Java Virtual Machine, macchina virtuale Java). Questa, naturalmente, funziona sempre in modalità MT anche in presenza del solo metodo main. Infatti, oltre all'applicazione utente, esistono altri T (definiti deamon, demoni) interamente gestiti dalla JVM, come per esempio il famoso GC (garbage collector, letteralmente raccoglitore di immondizia), il gestore della coda degli eventi grafici, etc.

Nelle tipiche implementazioni MT Java, i diversi T competono per aggiudicarsi le risorse condivise di cui hanno bisogno, incluse aree di memoria (heap) e quindi è necessario porre molta attenzione a problemi di sincronizzazione, di condivisione delle risorse, etc. Queste problematiche, che rappresentano una delle sfide principali nel disegno e implementazione di programmi MT, secondo la strategia iniziale di Java, potevano essere risolte utilizzando apposite parole chiavi come: syncronized e volatile, metodi quali, per esempio, le primitive: wait(), notify(), notifyAll(), contenuti nella classe antenata (java.lang.Object) da cui ereditano tutte le altre ed i metodi della classe java.lang.Thread. Questa strategia tuttavia presentava almeno tre importanti criticità:

Il disegno era fortemente basato sulla strategia del singolo monitor;

- l'utilizzo di questi strumenti faceva sì che molte applicazioni risultassero fragili: presenza di problemi di "race condition", "dead-lock" (caso classico dei metodi "java.lang.Thread: stop, suspend, resume, etc. ben preseto deprecati), e così via;
- si trattava di strumenti troppo a basso livello.

Strategia del singolo monitor

Nella quasi totalità delle applicazioni MT si verifica il caso in cui diversi T debbano contendersi l'accesso a risorse condivise e che quindi gli accessi debbano avvenire in mutua esclusione. A tal fine ogni linguaggio di programmazione fornisce appositi meccanismi atti a regolare l'accesso a tali risorse. La strategia tradizionalmente utilizzata dalla JVM consiste nell'associare un apposito oggetto **lock** (serratura) ad ogni classe ed istanza (figura 1). Da notare che in questo

contesto, con il termine di lock ci si riferisce al meccanismo interno di controllo dell'accesso agli oggetti e non alle primitive di lock introdotte con Java SE 5.0.

Le aree condivise da tutti i T sono lo heap e l'area dei metodi dove tutte le classi sono memorizzate. Quindi quando la JVM carica in memoria la definizione di una classe, vi associa immediatamente un'istanza (un oggetto) della classe java.lang.Class che, tra le altre responsabilità, si occupa anche di gestire il lock per l'accesso alla classe stessa.

I lock, in ogni istante di tempo, possono essere acquisiti da uno ed un solo T, pertanto quando un T acquisisce un lock nessun altro può accedere alla relativa area bloccata. Un T che intenda eseguire una porzione di codice ad accesso ristretto (sincronizzato), deve richiedere preventivamente l'acquisizione del relativo lock, quindi attenderne l'acquisizione, qualora un altro T ne sia in possesso. I lock è poi sbloccato automaticamente una volta che il T termina l'esecuzione delle istruzioni appartenenti all'area protetta.

In Java le primitive necessarie per l'acquisizione, il rilascio di questi lock, la verifica della disponibilità etc. sono trasparenti al programmatore il cui unico compito consiste nel definire le aree ad accesso ristretto. Ogni lock è gestito da un apposito oggetto denominato monitor, ciascun di questi si occupa di gestire uno ed un solo oggetto e si fa carico di sorvegliare le relative aree accesso esclusivo. Pertanto, qualora un T intenda entrare in un'area protetta (eseguirne la prima istruzione) di un oggetto, il relativo monitor si occupa di eseguire una serie

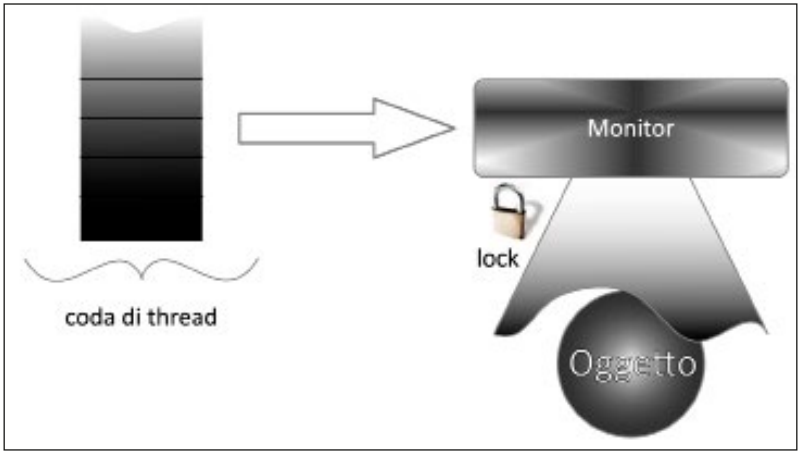


Figura 1. *Rappresentazione concettuale dei concetti di monitor e lock.*

di operazioni primitive necessarie per acquisirne il lock. Queste primitive, come vedremo di seguito, sono inserite, automaticamente, nel bytecode dal compilatore java. Uno stesso T può acquisire più volte il lock associato ad un medesimo oggetto (per esempio invocando un metodo ad accesso controllato di una determinata istanza che a sua volta ne invoca un altro sempre della stessa istanza, ancora ad accesso ristretto). Questa proprietà equivale a dire che lock Java sono rientranti. Si tratta di una caratteristica fondamentale per evitare che un T possa, autonomamente, generare una situazione paradossale di deadlock: rimanere i attesa di acquisire un lock già acquisito.

I monitor sono in grado di gestire queste situazioni utilizzando un apposito contatore che viene incrementato ogniqualvolta il lock viene acquisito e decrementato quando questo viene rilasciato.

I monitor, dunque, si occupano di sorvegliare zone ad accesso esclusivo e di avviare le operazioni necessarie per acquisire i lock. Tutto quello che deve fare il programmatore è definire queste zone ad accesso esclusivo. A tal fine è possibile utilizzare la parola chiave/costrutto synchronized. Le aree incluse nel costrutto sono definite sincronizzate.

Un T che intenda eseguire la parte di codice inclusa in un costrutto synchronized, come visto in precedenza, deve necessariamente acquisire il relativo lock, e quindi rilasciarlo all'uscita del costrutto. Queste operazioni richiedono, al livello di bytecode, l'esecuzione delle due primitive due primitive: monitorenter e monitorexit. Si tratta di istruzioni inserite dal compilatore java per delimitare, rispettivamente, l'inizio e la fine della parte sincronizzata.

Strumenti non in linea al modo di pensare applicazioni MT

Nonostante la presenza di meccanismi "nativi" per il supporto alla programmazione MT, la pratica insegna come la progettazione e realizzazione di applicazioni Java concorrenti, scalabili ed ad alte prestazioni, sia un'attività tutt'altro che semplice, in cui, frequentemente, gli sviluppatori finiscono per insabbiarsi e non sempre solo per proprie disattenzioni. La programmazione MT introduce nuove problematiche nel disegno delle applicazioni ed enfatizza quelle esistenti. Il disegno di avanzate applicazioni MT richiede uno studio attento a fattori quali: accesso a risorse condivise, comunicazione tra diversi flussi di esecuzione, performance, throughtput, parsimonioso utilizzo della memoria e delle risorse in generale, minimizzazione degli oggetti utilizzati, e così via. Inoltre, la progettazione di applicazioni MT Java, basate esclusivamente sulle primitive succitate, non è per nulla immediata, anzi è spesso e volentieri macchinosa al punto di indurre gli sviluppatori a cadere in una serie di insidie. Il problema di fondo è che

quando si progettano complesse applicazioni concorrenti, scalabili ad alte prestazioni, raramente si tende a pensare il sistema in termini di singoli T e di metodi quali **wait**() e **notify**(); primitive probabilmente troppo a basso livello. Risulta più naturale ragionare in termini di pool di T, semafori, mutex e barriere.

Logica conseguenza è che, in passato, il disegno di avanzate applicazioni MT Java, ha richiesto lo sviluppo preliminare e ad-hoc di complicati framework, atti a fornire un migliore livello di astrazione. L'obiettivo consisteva nel realizzare blocchi base, più o meno complessi, per il supporto della programmazione concorrente, con i soliti problemi derivati dalla cronica carenza di tempo: framework mai completamente compiuti, non sempre flessibili e riutilizzabili, faticosamente scalabili, carenti di documentazione, e così via. La situazione è drammaticamente cambiata grazie all'avvento del J2SE 5.0 (30 settembre del 2004 con la JSR 176) ed in particolare con l'introduzione del package della concorrenza (java.util.concurrent).

Problematiche classiche del MT

Il funzionamento tipico di applicazioni MT prevede che i vari T evolvano indipendentemente gli uni dagli altri e di tanto interagiscano, più o meno direttamente, per sincronizzarsi, per scambiare informazioni, per aggiornare dati memorizzati in aree condivise, e così via. Pertanto, molto frequente è il caso in cui diversi T debbano condividere appositi oggetti, e quindi aree di memoria heap. La condivisione di risorse se non progettata ed implementata correttamente, può dar luogo a diversi problemi piuttosto seri, spesso di difficile identificazione. I più noti sono: race conditions, deadlock, starvation e non deterministic behaviour.

Race conditions (condizioni di competizione)

Si tratta di un problema abbastanza ricorrente e si genera quando due o più T tentano di aggiornare contemporaneamente una medesima struttura di dati (oggetto) e, che, per via di un'implementazione non disegnata per un ambiente MT (tecnicamente non thread-safe), o non accurata, la lascino in uno stato inconsistente. Un esempio classico è quello di due T che aggiornano il valore di una medesimo contatore incapsulato in un apposito oggetto. Lo scenario tipico, in presenza di un codice non implementato accuratamente, prevede che il primo T acquisisca il valore del contatore (per esempio 933) e contemporaneamente o quasi, anche il secondo T legga il medesimo valore (ancora 933). A questo punto ciascun T dopo aver provveduto ad incrementare il contatore, memorizzi il nuovo valore nel contatore originario. Sebbene questo abbia subito due incrementi, il valore finale sarà comunque 934, e pertanto un incremento viene perso. Si immagini gli effetti

di tale situazione qualora il contatore rappresenti un sedile di un aeroplano, un posto al teatro o il prezzo e/o la disponibilità di un determinato prodotto finanziario.

Deadlock (abbraccio mortale)

Una tecnica spesso utilizzata per evitare problemi come quelli illustrati precedentemente (race conditions), consiste nel ricorrere ad appositi costrutti atti a far sì che l'accesso a determinate risorse condivise avvenga in mutua-esclusione. Pertanto, se un T accede ad una di queste risorse, gli altri automaticamente sono posti in attesa fintantoché il T stesso non rilasci la risorsa. Ciò fa sì che in presenza di una serie di strutture dati condivise da parte di più T, il relativo accesso sia regolato. Quindi i vari T, una volta accedute a una di tali strutture, automaticamente, la rende inaccessibile ad altri T. Qualora poi uno stesso T debba modificare diverse strutture può accadere che, per via di un'implementazione non furbissima, questo inizi a bloccare le varie risorse e ne trovi una o più bloccate da un altro T che, a sua volta, per evolvere abbia bisogno delle risorse bloccate dal primo T. A questo punto l'abbraccio mortale è generato con una serie di T che si bloccano vicendevolmente non riuscendo più ad evolvere. Sebbene, lo scenario presentato sia abbastanza semplice, in applicazioni complesse è possibile rigenerarlo in maniera abbastanza subdola.

Starvation (inedia)

L'inedia genera un risultato per molti versi simile a quello prodotto da un deadlock, in questo caso però le cause del blocco hanno una diversa connotazione. In particolare, si ha una situazione di inedia quando, sebbene in linea di principio uno o più T possano evolvere, praticamente, non lo fanno poiché non riescono ad ottenere una o più risorse necessarie per il proseguimento della loro esecuzione. La risorsa che più frequentemente può generare l'inedia di un T è indubbiamente la CPU. In questi casi un T non riesce a proseguire per via dell'impossibilità di aggiudicarsi cicli di CPU. Situazioni di inedia si possono generare in diversi modi, tra i quali, i più comuni sono il risultato di una inappropriata manipolazione della priorità di un insieme di T, della generazione di cicli infiniti da parte di un T che mantiene bloccata una risorsa condivisa, ecc.

Live-lock (blocco di vita)

Il live-lock è un particolare esempio di inedia che si genera nel caso in cui un T che sebbene non sia bloccato in attesa della disponibilità di una risorsa, non riesca a proseguire per la propria esecuzione in quanto impedito dalla necessità di ripetere l'esecuzione di un'istruzione che fallisca.

Non deterministic behaviour (comportamenti non deterministici)

Con questo termine ci si riferisce genericamente ad una serie di comportamenti inaspettati, anche molto seri, da parte di un'applicazione MT di cui però non si riesce ad identificarne l'esatta motivo. Race conditions, sono chiaramente, un esempio di comportamento non deterministico: ad un certo punto una o più variabili presentano dei valori errati. Se si considera che spesso il valore di determinate variabili influenzano il flusso del programma si comprende come quali conseguenze possano generarsi in applicazioni fortemente MT. Comportamenti non deterministici possono includere la produzione, in circostanze non chiare e quindi non facilmente riproducibili, di risultati inaspettati, calcoli errati, etc. Si pensi agli effetti che un comportamento non deterministico può avere in un sistema di calcolo automatico dei prezzi di offerta di determinate azioni finanziarie, op-pure in un sistema di controllo del traffico della metro, o in quello delle torri di controllo.

Chiaramente, un buon disegno ed implementazione costituiscono sempre il modo migliore per evitare problemi.

Il package della concorrenza

Il disegno di questo framework si deve essenzialmente a Doug Lea, professore universitario della State University of New York Oswego (lo stesso che ha dato luogo al framework delle collection), il quale, in tre anni di lavoro, è riuscito a mettere appunto un brillante framework Java per il supporto alla programmazione MT. Tale framework ha dimostrato livelli di performance e scalabilità così elevati, da meritare l'inclusione nella piattaforma Java 2 versione 5.0 ("Tiger"). Lo stesso Doug Lea, a cui si deve anche il framework delle collections Java, ha assolto il ruolo di specification leader della JSR 166.

Il package è costituito da una serie di classi che forniscono le fondamenta per la soluzione di un insieme di problemi appartenenti al particolare dominio delle applicazioni MT. Soluzioni concrete, ossia specifiche applicazioni MT, possono essere codificate implementando ben definite interfacce e/o estendendo specifiche classi e quindi combinando queste nuove classi con quelle esistenti. I principali meccanismi introdotti con il package della concorrenza sono:

- variabili atomiche
- schedulazione e gestione dell'esecuzione di task;
- strutture di dati "coda" e code bloccanti;
- nuove collezioni concorrenti (ottimizzate per il MT),
- blocchi (locks)
- sincronizzatori (semafori, barriere e variabili atomiche).

La libreria della concorrenza è organizzata in tre package: java.util.concurrent e i due sotto-package: locks e atomic.

Variabili atomiche (atomic variables)

Sono un insieme di classi (AtomicBoolean, AtomicInteger, AtomicIntegerArray, etc.) che permettono il trattamento atomico (non interrompibile) e, preferibilmente, non bloccante (il cui conseguimento dipende dai servizi offerti dalla piattaforma di esecuzione) delle variabili incapsulate. Si tratta di un concetto simile alle classi Java "wrapper" che incapsulano le variabili primitive Java (Boolean, Integer, etc.). In questo caso tuttavia l'obiettivo è disporre di classi ottimizzate per l'utilizzo in ambienti MT, e quindi in grado di offrire performance superiori a quelle che si otterrebbero con il ricorso al costrutto bloccante synchronized, grazie all'implementazione basata sulle primitive "compare and swap". Queste nuove istruzioni del tipo: boolean compareAndSet(expectedValue, updateValue), corrispondono all'implementazione, a basso livello, della strategia dell'optmistic locking: l'aggiornamento di un attributo avviene solo se il valore del parametro expectedValue coincide con quello reale. Qualora ciò si verifichi, il metodo ritorna il valore true, altrimenti false.

Locks e Conditions

Come visto in precedenza, tradizionalmente, il linguaggio Java gestiva i lock in maniera del tutto trasparente. Con l'introduzione del package della concorrenza si è introdotta la possibilità di una gestione esplicita. Gli strumenti lock (serrature) forniscono avanzati meccanismi per controllare l'accesso a risorse condivise (logiche e/o fisiche) da parte di diversi thread. **Si tratta di una sofisticata alternativa al costrutto Java synchronized**, la cui gestione basata sul singolo monitor raramente risulta essere ottimale in ambienti MT sia per problemi di performance, sia di limitato controllo. Il meccanismo dei lock, oltre a presentare migliori prestazioni, fornisce una serie di utilissimi servizi aggiuntivi, quali:

- la possibilità di interrompere un T in attesa di uno specifico lock;
- la dichiarazione del tempo massimo di attesa di un T per uno specifico lock;
- la generazione di insiemi di meccanismi "wait-notify" (chiamati condition) associati ad uno stesso lock.

Le classi del package della concorrenza, utilizzano sistematicamente il meccanismo dei lock ed in particolare la classe ReentrantLock, al posto del costrutto synchronized. L'unico effetto collaterale dovuto all'utilizzo di questo meccanismo, consiste nel dover codificare esplicitamente l'acquisizione (myLock.lock()) ed il rilascio (myLock.unlock()) dei lock che, ovviamente, richiede la presenza

del costrutto try...finally onde evitare l'insorgere di situazioni di dead-lock in situazione di eccezione.

Sincronizzatori (semaphore, barrier, latch e Exchanger)

Tutti coloro che hanno studiato la programmazione concorrente, ricorderanno il concetto dei semafori di Dijkstra. Questi sono stati (finalmente) introdotti con il package java.util.concurrent, attraverso la classe Semaphore. Si tratta di un ennesimo meccanismo utilizzato per limitare l'accesso a risorse condivise (tipicamente ad un pool), logiche e/o fisiche che siano, da parte di un predefinito numero di thread. I semafori, detti anche semafori contatori, consistono in un insieme di permessi che possono essere ottenuti (acquire()) finché c'è ne è uno disponibile (il contatore interno è maggiore di 0). La relativa restituzione (release()) genera l'incremento del contatore, e, potenzialmente, l'uscita dallo stato di attesa da parte di un T che precedentemente aveva richiesto di eseguire l'acquisizione di un permesso. Il concetto della barriera, introdotto per mezzo della classe CyclicBarrier, che fornisce un meccanismo molto conveniente per la sincronizzazione di un insieme di thread. In particolare, come suggerisce il nome, offre la possibilità di specificare nel codice dei punti (myBarrier.await()) in cui i vari T sono forzati ad attendere che i restanti raggiungano quel medesimo punto prima di poter proseguire oltre. Se, per qualche ragione, un T in attesa lascia prematuramente la barriera stessa (per esempio scade il relativo tempo massimo di attesa, timeout), l'oggetto barriera transita in uno stato di rottura (broken) che forza i restanti a procedere oltre. Il nome ciclico (cyclic) deriva dal fatto che un medesimo oggetto barriera può essere utilizzato diverse volte tramite l'invocazione del metodo reset(). Anche il meccanismo delle barriere è utile per decomporre calcoli particolarmente complessi in un insieme di sottocalcoli di complessità inferiore (disegno "divide et impera").

Altro meccanismo simile alla barriera è costituito dalla classe CountDownLatch (serratura a scatto a decremento), la quale fa sì che un T rimanga in attesa dei risultati forniti da opportune operazioni la cui completa esecuzione è assegnata ad altri thread. Istanze di questo tipo prevedono come parametro del metodo costruttore un valore di inizializzazione del contatore interno. Il corrispondente metodo di attesa (myLatch.await()) blocca il T chiamante finché il contatore interno raggiunge il valore 0. Il decremento avviene quando gli altri thread, ai quali va fornito il riferimento dell'istanza CountDownLatch (in questo caso myLatch), invocano esplicitamente l'operazione di decremento (myLatch.countdown()). L'ultima classe presente, appartenente a questa famiglia è Exchanger<V>, si tratta di un potente meccanismo che consente di definire

punti di incontro (rendezvous) tra coppie di thread, con esplicito scambio di dati. In particolare, la primitiva exchange, fa sì che il T invocante attenda, sia bloccato nell'esecuzione del metodo, finché un altro T giunga al medesimo punto e quindi fornisca il riferimento all'oggetto di scambio dichiarato. Anche questo metodo prevede la possibilità di definire il tempo massimo di attesa di un thread.

Strutture dati (Collection) MT

Quantunque le collezioni standard Java siano thread-safe, nativamente quelle di Java 1 (Hashtable, Vector, etc.) o su richiesta quelle introdotte con Java 2 (Collections.synchronizedMap() , Collections.synchronizedList(), etc.), queste non sempre risultano particolarmente efficienti e scalabili per applicazioni MT. Il relativo utilizzo in questi ambienti finisce frequentemente per generare odiosi colli di bottiglia, dovuti principalmente alla strategia del singolo monitor. Il nuovo package della concorrenza risolve questa limitazione fornendo nuove classi che implementano strutture dati di tipo lista, set e hashtable, rispettivamente: CopyOnWriteArrayList, CopyOnWriteArraySet e ConcurrentHashMap, disegnate specificatamente per ambienti MT. Quindi presentano comportamenti ottimizzati per scenari in cui diversi thread, "contemporaneamente", necessitano di leggere e/o scrivere su istanze di tali classi.

Altra variazione importante consiste nella definizione dell'interfaccia Queue, correttamente inserita nel package java.util, disegnata per definire il comportamento di oggetti code. **L'interfaccia Queue, come lecito attendersi, è stata disegnata per un utilizzo generale e quindi non contiene metodi specifici per il supporto al MT**. Questi, necessari per la soluzione di scenari del tipo produttore/consumatore, sono invece presenti nell'interfaccia BlockingQueue che presenta tutta una serie di implementazioni come: ArrayBlockingQueue, DelayQueue, LinkedBlockingQueue, PriorityBlockingQueue e SynchronousQueue inserite nel package java.util.concurrent.

Executor e lo scheduling ed esecuzione di task.

La stragrande maggioranza di applicazioni MT richiedono l'implementazione di pool di T e la presenza di soluzioni a problematiche quali lo scheduling dei task e il controllo della relativa evoluzione. Il ricorso a pool di T è un'ottima strategia sia per migliorare le performance delle applicazioni MT (disponendo di T pronti ad essere utilizzati, si elimina la latenza richiesta dalla creazione di un nuovo thread), sia per controllarne la quantità, onde evitare la presenza di un numero eccessivo di T finisca per generare l'effetto contrario: la diminuzione delle

performance. Il pool di thread, inoltre, tende a semplificare il problema dell'assegnazione delle risorse ai task: una strategia spesso utilizzata consiste nel pre-assegnare ai thread, all'atto della relativa creazione, le risorse necessarie per l'esecuzione dei task. Le interfacce e classi che implementano i meccanismi base di quest'area sono:

- esecutori (Executor, ExecutorService, ScheduledExecutorService, etc.);
- esecutori basati su pool di T (ScheduledThreadPoolExecutor e ThreadPoolExecutor);
- operazioni a termine (Future e FutureTask).

In particolare, il framework degli esecutori fornisce una serie di meccanismi decisamente flessibili per la gestione dei task. Questi sono rappresentati da classi che implementano l'interfaccia Runnable, che quindi definiscono il corpo del metodo run(), eseguibile da opportuni thread.

Il package dispone di meccanismi che permettono di effettuare: l'esecuzione asincrona, oltre che ovviamente quella sincrona, la richiesta di informazioni, il controllo dell'esecuzione ed eventualmente la cancellazione di task, la cui gestione avviene in funzione di specifiche politiche, dichiarabili separatamente. Il package dispone poi di una serie di pool di T assolutamente flessibili, i quali possono essere generati sia:

- direttamente tramite le classi: ThreadPoolExecutor, ScheduledThreadPoolExectutor, la quale permette di gestire task i cui comandi possono essere eseguiti dopo un determinato intervallo di tempo e/o periodicamente;
- utilizzando appositi meccanismi di factory (Executors.newCachedThreadPool(), Executors.newFixedThreadPool(int) e Executors.newSingleThreadExecutor()).

Le diverse implementazioni sono sempre isolate da opportune interfacce, le quali, tra l'altro permettono di definire meccanismi standard per la terminazione (shut-down) controllata degli esecutori e per la gestione della cancellazione di lavori non ancora eseguiti. La descrizione di quest'area del MT è però demandata al prossimo articolo..

Conclusioni

Questo capitolo presenta una breve descrizione dell'evoluzione della strategia MT in Java la ricapitolazione di alcune feature del package java.util.concurrent, necessarie a una piena comprensione delle nuove feature Java SE 7 illustrate nel Capitolo 3. Queste continuano il percorso di revisione della strategia Java per il supporto al MT iniziato con J2SE 5.0. Sebbene, Java sia stato datato fin dalle versione iniziali, di un supporto nativo del MT, che almeno inizialmente sembrava

veramente un'ottima idea in discontinuità con i linguaggi del passato, ben presto tutte le sue limitazioni sono divenute evidenti: eccessiva rigidità (singolo monitor), generazione di colli di bottiglia (i meccanismi di controllo della concorrenza erano eccessivamente coarse-grained), feature troppo a basso livello (Thread, synchronised, wait and notify), ridotto controllo, etc. Ora, con il senno di poi è possibile affermare che la strategia iniziale del MT di Java fu più una limitazione che un reale vantaggio.

Appendice B
New Input-Output API

Prefazione

In questa appendice presentiamo brevemente il framework NIO (New Input/Output, Nuova libreria di Input/Output). L'obiettivo non è quello di fornirne una descrizione dettagliata, bensì di presentare i meccanismi base la cui conoscenza è propedeutica ad una piena comprensione del Capitolo 4 "NIO2".

Breve storia di NIO

La prima versione delle API NIO fu introdotta con la versione Java J2SE 1.4 per mezzo della JSR51. J2SE 1.4 fu rilasciata il 6 febbraio del 2002 con il nome di Merlin (Smeriglio, piccolo falco in cui nome scientifico è Falco columbarius, per via della sua preda preferita: i colombi). L'API NIO venne introdotta per sopperire ad alcune lacune evidenziate dal package I/O originale di Java (java.io) implementato fin dalla versione Java 1.0. Sebbene per molti programmatori l'API Java NIO sia stata introdotta essenzialmente per disporre di operazioni di I/O non bloccanti, in realtà è stato fatto molto di più a partire dall'introduzione di un nuovo ed integrato elegante disegno. Le principali feature introdotte furono:

- possibilità di eseguire il mapping in memoria dei file;
- operazioni orientate ai blocchi (al posto del tradizione flusso di byte) che sfruttano i servizi messi a disposizione dalle piattaforme di esecuzione;
- operazioni di I/O asincrone e quindi non bloccanti e lock dei file o di opportune sezioni.

Il package java.nio include molti elementi interessanti, tuttavia il framework è basato su tre concetti fondamentali (figura 1):

- buffer (contenitori);
- channel (canale);
- selector (selettore).

Buffer

I buffer che, come lecito attendersi, sono contenitori di dati. java.nio.Buffer è una classa astratta che prevede una serie di specializzazioni: ByteBuffer,

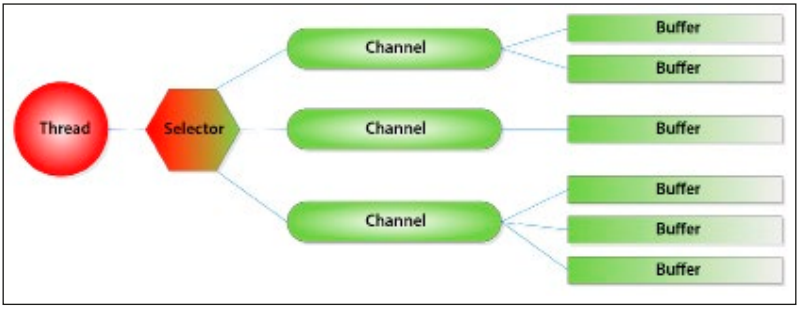


Figura 1. *Schema logico degli elementi fondamentali dell'architettura dell'API NIO.*

CharBuffer, DoubleBuffer, FloatBuffer, ecc. ognuna dotata di rispettivi decoder ed encoder che eseguono le varie trasformazioni, come per esempio la classe astratta java.nio.charset.CharsetEncoder le cui specializzazioni sono in grado di tradurre sequenze di caratteri di 16bit Unicode nella corrispondente rappresentazione a byte. Ovviamente, le specializzazione della classe astratta CharsetDecoder eseguono esattamente l'opposto. Il vantaggio nel disporre di buffer risiede sia nella maggiore efficienza (l'API effettua il cashing dei dati), sia nella possibilità di eseguire una serie di operazioni che non sarebbero altresì possibili automaticamente come per esempio muoversi avanti ed indietro nella lettura de dati. I lettori più attenti potrebbero obiettare che anche la libreria di IO standard include la possibilità di eseguire operazioni bufferizzate (per esempio BufferedReader), tuttavia in questo caso esistono ancora i vari limiti della libreria di IO spiegati di seguito (operazioni bloccanti) ed inoltre oggetti BufferedReader non possiedono la flessibilità ed il controllo degli oggetti Buffer.

Channel

I channel ossia canali di vario tipo che rappresentano le connessioni con dispositivi di vario tipo in grado di eseguire operazioni I/O. In parole semplici sono un po' l'equivalente degli stream: oggetti dal quale si può leggere e/o scrivere dati attraverso i buffer. Un'importante caratteristica da ricordare è che mentre gli stream sono mono-direzionali (le varie classi sono specializzazione o della casse astratta InputStream o della classe astratta OutputStream), i canali invece sono bi-direzionali, il che li rende più simili al funzionamento delle varie piattaforme sottostanti (Sistemi Operativi).

Selector

I selectors (selettori), che insieme ai canali permettono di definire operazioni multiplex e non-bloccanti. Multiplex significa che uno stresso Thread è in grado di gestire diversi canali. In particolare, la libreria NIO rende possibile registrare diversi canali ad un unico selettore (figura 1). Ciò permette di utilizzare un singolo Thread per selezionare i canali che dispongono di input pronto ad essere processato o i canali pronti a scrivere. Pertanto, il meccanismo dei selettori semplifica la gestione di diversi canali da parte di un singolo Thead. Operazioni non-bloccanti significa che quando un Thread richiede dei dati da un canale, questa operazione ritorna immediatamente fornendo i dati correntemente disponibili (eventualmente nulla) senza rimanere bloccato fino al memento in cui ci siano dei dati disponibili per la lettura. Ciò lascia il Thread libero di eseguire altri compiti come per esempio eseguire altre operazioni di I/O su diversi canali. Chiaramente, lo stesso comportamento accade quando un Thread richiede di scrivere dei dati e non deve attendere che questi sia completamente scritti.

Facendo un paragone tra Java I/O e Java NIO, le differenze fondamentali sono le seguenti:

- Java I/O è orientato allo stream (**stream oriented**) e prevede operazioni bloccanti (**blocking I/O**);
- Java NIO è orientato al buffer (**buffer oriented**), prevede operazioni non bloccanti (**non blocking I/O**) e supporto di selettori (**Selectors**).

Un esempio

L'esempio mostrato è un classico Server NIO/Client NIO, dove quest'ultimo è a sua volta un server come mostrato nel codice riportato di seguito. Visti i fini di questo capitolo, il client è stato implementato come un test di unità (JUnit).

Il funzionamento del server è molto semplice e prevede che una volta lanciato resti in attesa di collegamenti da parte di client esterni (per essere precisi alterna intervalli in attese non bloccanti al controllo dello stato/richieste di shutdown). Una volta stabilita una connessione con un cliente, attende che questo invii il proprio id ("CLIENT_ID:") seguti dal codice di richiesta del servizio ("TS_REQ."). Alla ricezione di questa sequenza, il server risponde al client riportando il corrispondente id seguito dalla risposta del servizio ("TS_RESP:"): data ed orario corrente (new Date()).

Il tutto è mostrato nel sequence diagram di figura 2.

```
public class SimpleServer implements Runnable {
```

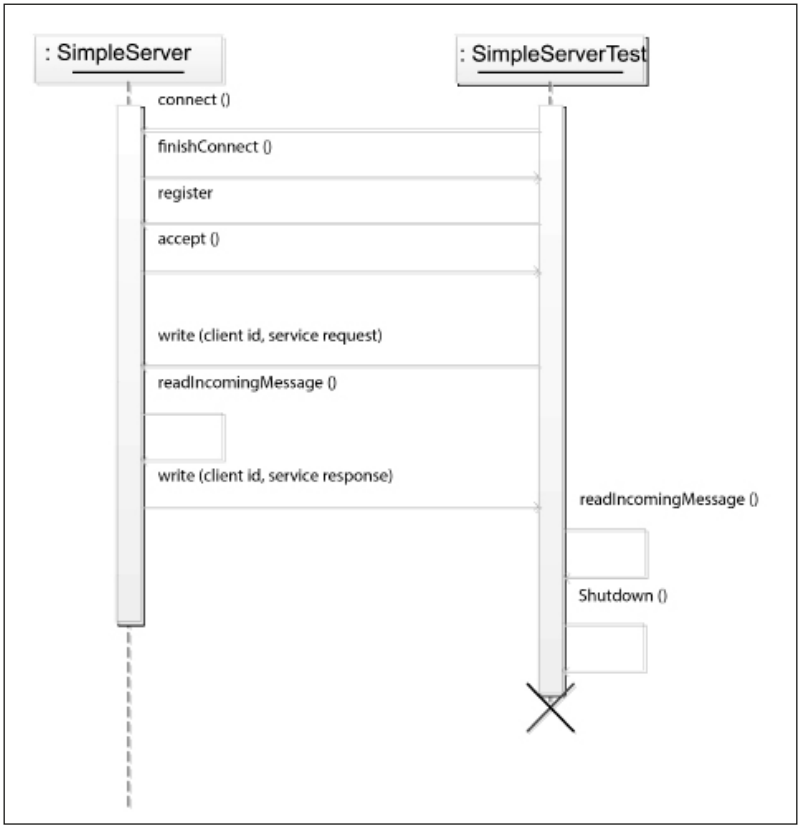


Figura 2. *Sequence diagram della comunicazione client server.*

```
// -------------- CONSTANTS SECTION  --------------

/** logger */
private static Logger LOGGER =
  Logger.getLogger(SimpleServer.class);

/** default message size */
private static short MESSAGE_SIZE = 1024;
```

```
  /** default port */
  private static final int DEFAULT_PORT = 8900;

  /** time window to exit from the wait for new stimuli */
  private static final int CHECK_SHUTDOWN_INTERVAL = 200;

  /** Charset */
  private static final Charset charset = Charset.forName("UTF-8");

  /** Charset decoder for UTF-8 */
  private static final CharsetDecoder decoder = charset.
newDecoder();

  /** Client time service request */
  public static final String CLIENT_TS_REQ  = " TS_REQ.";
  /** time service response */
  public static final String CLIENT_TS_RESP = " TS_RESP:";
  /** client identification string */
  public static final String CLIENT_ID   = "CLIENT_ID:";

  // --------------- ATTRIBUTES SECTION  ---------------

  /** server statuses */
  private enum ServerState
     {STARTED, RUNNING, SHUTTING, OFF  };

  /** current status */
  private ServerState currentState =  ServerState.OFF;

  /** server port, this is used to accept requests */
  private int port = DEFAULT_PORT;

  /** sever selector */
  private Selector selector = null;

  /** sever socket channel */
  private ServerSocketChannel serverSocket = null;
```

```
/** direct byte buffer for reading */
private static ByteBuffer readBuffer =
   ByteBuffer.allocateDirect(MESSAGE_SIZE);

// --------------- METHODS SECTION  ---------------


// -------------------------------- constructors

/**
 * Default constructor
 * Uses the default port
 */
public SimpleServer() {
}

/**
 * Construct
 * @param port    server port
 */
protected SimpleServer(int port) {
  this.port = port;
}

// ------------------------------ getters/setters

/**
 * Get the port on which this server accepts connections
 * @return the server port
 */
public int getPort() {
  return port;
}

/**
 * Returns the server host
 * @return server host
 */
```

```
public InetAddress getServer() {

  InetAddress inetAddress = null;

  try {

    inetAddress = InetAddress.getLocalHost();

  } catch (UnknownHostException uhe) {
    LOGGER.warn("EXCEPTION: "+uhe.getMessage());
  }

  return inetAddress;
}


/**
 * Set the server state
 * @param newState   the new state to set
 */
private synchronized void setServerState(ServerState newState) {
  this.currentState = newState;
}


/**
 * @return true if this server has been started
 *    false otherwise
 */
public boolean isStarted() {
  return ServerState.STARTED.equals(currentState);
}


/**
 * @return true if this server is currently running
 *    false otherwise
 */
```

```
public boolean isRunning() {
  return ServerState.RUNNING.equals(currentState);
}

/**
 * @return true  if this server is currently in off state
 *    false otherwise
 */
public boolean isOff() {
  return ServerState.OFF.equals(currentState);
}

// ---------------------------------------- server methods

/**
 * Initialise the server
 * @throws IOException in case a serious problem occurs
 */
protected void init() throws IOException {

  setServerState(ServerState.STARTED);

  selector = Selector.open();

  serverSocket = ServerSocketChannel.open();
  serverSocket.socket().bind(new InetSocketAddress(port));
  serverSocket.configureBlocking(false);
  serverSocket.register(selector, SelectionKey.OP_ACCEPT);

  setServerState(ServerState.RUNNING);
}

/**
 * Shutdown the sever
 */
protected void shutDown() {

  setServerState(ServerState.SHUTTING);
```

```
    LOGGER.info("Shutting down the server");

    if (selector != null) {
      try {
        selector.close();
      } catch (IOException e) {
        // ignore this exception
      }
    }

    if (serverSocket != null) {

      if (serverSocket.socket() != null) {
        try {
          serverSocket.socket().close();
        } catch (IOException e) {
          // ignore this
        }
      }

      if (serverSocket != null) {
        try {
          serverSocket.close();
        } catch (IOException e) {
          // ignore this
        }
      }
    }

    setServerState(ServerState.OFF);
  }


  /**
   * Run the server
   *
   * @throws
   */
```

```
public void run() {

  LOGGER.info("Starting the server on"+
       " host:"+this.getServer()+
       " port:"+this.getPort());

  if ( ServerState.RUNNING.equals(currentState)) {
    LOGGER.warn("The server is already running.");
  }

  try {
    init();

    // run the sever as long as its status is set to running
    while (ServerState.RUNNING.equals(currentState) ) {

      // check every x ms whether the server
      // has been requested to stop
      selector.select(CHECK_SHUTDOWN_INTERVAL);
      LOGGER.info("Out of the select wait…");

      Iterator<SelectionKey> allSelectionKeys =
         selector.selectedKeys().iterator();

      while ( allSelectionKeys.hasNext()) {

        SelectionKey currKey = allSelectionKeys.next();
        allSelectionKeys.remove();

        try {

          LOGGER.info(
             "Current selection Key. "+
             " connectable:"+currKey.isConnectable()+
             " acceptable:"+currKey.isAcceptable()+
             " readable:"+currKey.isReadable()+
             " writeable:"+currKey.isWritable());
```

```
if (currKey.isConnectable()) {
  LOGGER.info("Try to connect…");

  SocketChannel socketChannel =
     (SocketChannel)currKey.channel();
  socketChannel.finishConnect();

  LOGGER.info("Conneted? "+
     socketChannel.isConnected());
}

if (currKey.isAcceptable()) {
  LOGGER.info("Try to accept connection…");

  SocketChannel client = serverSocket.accept();
  client.configureBlocking(false);
  client.socket().setTcpNoDelay(true);
  client.register(selector,
        SelectionKey.OP_READ |
        SelectionKey.OP_WRITE);

  LOGGER.info("Registered? "+client.isRegistered());
}

if (currKey.isReadable()) {
  LOGGER.info("Try to read…");
  CharBuffer charBuffer =
       readIncomingMessage(currKey);
  currKey.cancel();

  if (charBuffer != null) {
    // Respond with the current time
    String receivedMsg =charBuffer.toString();
    if (isValidRequest(receivedMsg) ) {

      String response =
         receivedMsg.replaceFirst(
```

```
                    SimpleServer.CLIENT_TS_REQ,
                    SimpleServer.CLIENT_TS_RESP);
              response += new Date();
              LOGGER.info("Message To send:" + response);
              writeMessage(currKey, response.getBytes());
            }
           }
         }

        } catch (IOException ioe) {
          LOGGER.warn("EXCEPTION :"+ioe.getMessage());
          resetKey(currKey);
        }
       }
     }

     LOGGER.info("Server no longer running");

   } catch (IOException ioe) {
     LOGGER.warn("EXCEPTION :"+ioe.getMessage());

   } finally {
     shutDown();
   }
}


/**
 * Verify whether the given message include a valid
 * client request
 * @param msg   message to analyse
 * @return true it is a valid request
 *    false otherwise
 */
protected boolean isValidRequest(String msg) {
  return (msg.indexOf(SimpleServer.CLIENT_ID) != -1) &&
       (msg.indexOf(SimpleServer.CLIENT_TS_REQ) != -1);
}
```

```
/**
 * Request the server to stop in a graceful manner.
 * @return true if the server was successfully set to stop.
 */
public void requestToShutdown() {
  LOGGER.info("Request to shut-down");

  setServerState(ServerState.SHUTTING);
}


/**
 * Remove any internal state for the given key.
 * @param key the key to be cancelled.
 */
protected void resetKey(SelectionKey key) {
  key.cancel();
}


/**
 * Send the given message to the specified client.
 *
 * @param buffer the message to send.
 */
protected void writeMessage(
            SelectionKey channelKey,
            byte[] buffer)  {

  LOGGER.info("Try to send a message…");

  if (buffer == null) {
    return;
  }

  // copying into byte buffer
  ByteBuffer writeBuffer =
      ByteBuffer.allocate(buffer.length);
```

```
    writeBuffer.put(buffer);
    writeBuffer.flip();

    if ( (buffer != null) &&
       (ServerState.RUNNING.equals(currentState)) ) {

      int numWrittenBytes;

      try {
         // only 1 thread can write to a channel at a time
        SocketChannel channel =
           (SocketChannel)channelKey.channel();

        synchronized (channel) {
          numWrittenBytes = channel.write(writeBuffer);
        }

        if (numWrittenBytes == -1) {
          resetKey(channelKey);
        }

      } catch (IOException ioe) {
        LOGGER.warn("Exception :"+ioe.getMessage());
        resetKey(channelKey);
      }
    }
  }


  /**
   * Read incoming messages.
   *
   * N.B. Read the time from the client
   *    For simplicity it is assumed that the message
   *    can be stored into a single buffer.
   *    Hence the server needs to read it only once
   *
   * @param selectionKey the client connection to read
```

```
     * @return the messages read as a char buffer
     * @throws IOException if the client connection is closed
     */
    private CharBuffer   readIncomingMessage(
                  SelectionKey selectionKey)
        throws    IOException {

      readBuffer.clear();

      ReadableByteChannel readableChannel =
         (ReadableByteChannel)selectionKey.channel();

      if (readableChannel.read(readBuffer) == -1) {
        throw new IOException("try to read on closed key.");
      }

      readBuffer.flip();

      CharBuffer messageBuffer =
                decoder.decode(readBuffer);

      LOGGER.info("Message Received:" +
                messageBuffer.toString());

      return messageBuffer;
    }

}
```

Listato 1. *Codice del server NIO.*

Il codice mostrato nel listato 1 dovrebbe essere auto esplicativo, pertanto nelle righe seguenti riportiamo solo alcune informazioni relative alle parti più strettamente legate al framework NIO.

Il server una volta avviato si occupa di eseguire la propria inizializzazione: invoca il metodo init, il quale, per prima cosa, esegue l'operazione di creazione di un selettore: **selector = Selector.open**(). Come visto in precedenza, i selettori

sono elementi centrali per le comunicazioni del framework NIO: sono gli oggetti in cui è possibile registrare l'interesse a ricevere determinati eventi I/O, registrazione che poi gli permette di notificare l'occorrenza di tali eventi.

Il passo successivo consiste nel creare un canale per ognuna delle porte in cui il server intende ricevere comunicazioni. Nel caso in questione il server opera su un'unica porta, quindi il codice include le seguenti istruzioni: **serverSocket = ServerSocketChannel.open()** per la creazione del canale, **serverSocket.configureBlocking(false)** per impostarlo in modalità non bloccante, **serverSocket.socket().bind(new InetSocketAddress(port))** per eseguire il binding con la porta. A questo punto si è pronti a registrare il selettore sul nuovo canale: **serverSocket.register(selector, SelectionKey.OP_ACCEPT)**. Da notare che il secondo parametro specifica che il server è interessato a ricevere eventi che hanno luogo quando viene stabilita una connessione con il client (più avanti incontreremo altri eventi). Con questa operazione termina l'inizializzazione del server e quindi questo è pronto ad operare eseguendo un loop (while) che termina soltanto quando il server riceve una richiesta di shutdown o transita in una condizione di errore (viene generata una eccezione).

All'interno del loop, il controllo rimane in attesa di ricevere eventi esterni **selector.select(CHECK_SHUTDOWN_INTERVAL)**. Per la precisione, non si tratta di un'attesa indefinita ma temporeggiata: anche in assenza di stimoli esterni, esce dal select dopo un intervallo di tempo pari a 200 millisecondi. Questi sblocchi sono necessari per verificare se nel frattempo sia avvenuta una richiesta di shutdown.

Qualora **select** termini in presenza di eventi, restituisce il numero di eventi ricevuti. L'istruzione successiva consiste nell'ottenere un oggetto iteratore di questi eventi per poterli gestire: **Iterator<SelectionKey> allSelectionKeys = selector.selectedKeys().iterator()**. Per ciascun evento (**SelectionKey currKey = allSelectionKeys.next()**) è necessario comprendere la tipologia e gli elementi coinvolti. Il nostro server si occupa di gestire eventi: connectable (**isConnectable()**), acceptable (**isAcceptable()**) e readeable (**isReadeable()**).

Nel caso ci sia un evento di richiesta connessione, il server esegue le seguenti istruzioni: **SocketChannel socketChannel = (SocketChannel)currKey.channel()** e **socketChannel.finishConnect()**. Ciò fa sì che la comunicazioni si instauri e si generi l'accettazione della comunicazione con relativo evento. La gestione di quest'ultimo richiede i seguenti passi:

```
SocketChannel client = serverSocket.accept();
client.configureBlocking(false);
```

```
client.register(selector,
                SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```

Si accetta la comunicazione, si importa il canale in modalità non bloccante e si specifica che si è interessati a leggere e scrivere.

A questo punto la rimanente parte del codice non dovrebbe presentare alcun segreto e quindi la relativa comprensione viene lasciata come esercizio per il lettore.

Di seguito è riportato il codice della classe di test del server implementata attraverso il framework JUnit. In realtà anche il client implementa un semplice server che come tale non dovrebbe più rappresentare un problema.

```
public class SimpleServerTest {

  // -------------- CONSTANTS SECTION  --------------

  /** logger */
  private static Logger LOGGER =
     Logger.getLogger(SimpleServerTest.class);

  /** instance of the running sever */
  private static SimpleServer server;

  // --------------- ATTRIBUTES SECTION  -------------

  // --------------- METHODS SECTION  ---------------

  /**
   * Waste time. Necessary to allow the dust to settle
   * @param timeToWaste  time to waste
   */
  private static void wasteTime(long timeToWaste) {
    try {
      Thread.sleep(timeToWaste);
    } catch (InterruptedException e) {
      // ignore this
    }
  }
```

```
@BeforeClass  // start the server
public static   void   setup()
          throws IOException {

  LOGGER.info(“>TEST>Setting up the test”);
  server = new SimpleServer();
  new Thread(server).start();

  wasteTime(100);

  assertTrue(server.isRunning());
}


@AfterClass  // shut-down the server
public static   void   tearDown()
          throws IOException {

  LOGGER.info(“>TEST>Closing the test”);

  wasteTime(100);

  server.requestToShutdown();

  wasteTime(200);

  assertTrue(server.isOff());
}


/**
 * Connect to the server
 * 1. Open the channel
 * 2. set it to non-blocking
 * 3. initiate connect
 *
 * @return newly created socket channel
 */
```

```
private SocketChannel connectToServer() {
  SocketChannel socketChannel = null;

  InetSocketAddress socketAddress
    = new InetSocketAddress(
      server.getServer(),
      server.getPort());

  try {

    LOGGER.info(“>TEST>Connecting to the server”);
    socketChannel = SocketChannel.open();
    socketChannel.configureBlocking(false);

    socketChannel.connect(socketAddress);

    socketChannel.finishConnect();

    assertTrue(socketChannel.isConnected());

    LOGGER.info(“>TEST>Connected”);

  } catch (IOException ioe) {
    LOGGER.warn(“>TEST>Exception:”+ioe);
    closeChannel(socketChannel);
    fail();
  }

  return socketChannel;
}


@Test
public void testReceiveAndSendMessage()  {

  SocketChannel socketChannel = connectToServer();

  try {
```

```
LOGGER.info(“>TEST>Registering…”);
Selector selector = Selector.open();
socketChannel.register(
   selector,
   SelectionKey.OP_READ | SelectionKey.OP_WRITE);

// indicate whether the client
// has written to the sever
boolean wroteRequest = false;
// stop the client
boolean stop = false;

while ( (selector.select() > 0) && (!stop) ) {

  Iterator<SelectionKey> allKeys =
     selector.selectedKeys().iterator();

  while (allKeys.hasNext()) {

    SelectionKey key = allKeys.next();
    allKeys.remove();

     LOGGER.info(“>TEST>Selection Key. “+
         “ connectable:”+key.isConnectable()+
         “ acceptable:”+key.isAcceptable()+
         “ readable:”+key.isReadable()+
         “ writeable:”+key.isWritable());

    if (key.isReadable()) {
      readIncomingMessage(key);
      stop = true;
    }


    if (key.isWritable()) {

      if (!wroteRequest) {
        writeMessage(socketChannel);
```

```
          }

          wroteRequest = true;
        }
      }
    }

    assertTrue(wroteRequest);
    assertTrue(stop);

  } catch (IOException ioe) {
    fail();

  } finally {
    closeChannel(socketChannel);
  }
}


/**
 * Write a message to the server
 * @param socketChannel  socketChannel to use
 * @throws IOException a serious problem occurred
 */
Private void   writeMessage(SocketChannel socketChannel)
     Throws IOException {

  LOGGER.info(“>TEST>Writing message…”);

  // ---- prepare the message and send it
  String clientId =
     SimpleServer.CLIENT_ID+
     “Test Client”;
  String msg =
     clientId+
     SimpleServer.CLIENT_TS_REQ;

  ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());
```

```
  socketChannel.write(buffer);

  LOGGER.info(“>TEST>Wrote message.”);
}


/**
 * Read the incoming message from the server
 * @param key selection key to use to read the message
 * @throws IOEXception if a serious problem occurred
 */
Private void  readIncomingMessage(SelectionKey key)
     throws IOException {

  LOGGER.info(“>TEST>Reading incoming message…”);

  ByteBuffer readBuffer = ByteBuffer.allocate(1024);
  readBuffer.clear();

  ReadableByteChannel readableChannel =
     (ReadableByteChannel)key.channel();

  if (readableChannel.read(readBuffer) == -1) {
    throw new IOException(“try to read on closed key. “);
  }

  readBuffer.flip();

  CharsetDecoder decoder =
     Charset.forName(“UTF-8”).newDecoder();

  CharBuffer messageBuffer = decoder.decode(readBuffer);

  assertNotNull(messageBuffer);

  String msg = messageBuffer.toString();

  assertTrue(msg.indexOf(SimpleServer.CLIENT_ID) != -1);
```

```
    assertTrue(msg.indexOf(SimpleServer.CLIENT_TS_RESP) != -1);

    LOGGER.info(“>TEST>Read message:”+messageBuffer.toString());
  }


  /**
   * Close the given channel
   * @param socketChannel   channel to close
   */
  private void closeChannel(SocketChannel socketChannel) {

    LOGGER.warn(“>TEST>Closing channel”);

    if (socketChannel != null) {
      try {
        socketChannel.close();
      } catch (IOException e) {
        // ignore this exception
      }
    }
  }
}
```

Listato 2. *Il server test, ossia il client NIO.*

Di seguito è riportato un esempio dell'output generato, privato delle informazioni strettamente legate al logging. Da notare che le linee generate dalla classe di test sono premesse dal prefisso “**TEST>**”.

```
- >TEST>Setting up the test
- Starting the server on host:XXXXXXX/169.60.125.194 port:8900
- >TEST>Connecting to the server
- >TEST>Connected
- >TEST>Registering…
- Out of the select wait…
- Current selection Key.  connectable:false acceptable:true
```

```
readable:false writeable:false
- Try to accept connection…
- Registered? true
- Out of the select wait…
- Current selection Key.  connectable:false acceptable:false
readable:false writeable:true
- >TEST>Selection Key.  connectable:false acceptable:false
readable:false writeable:true
- Out of the select wait…
- >TEST>Writing message…
- Current selection Key.  connectable:false acceptable:false
readable:false writeable:true
- >TEST>Wrote message.
- Out of the select wait…
- >TEST>Selection Key.  connectable:false acceptable:false
readable:false writeable:true
- Current selection Key.  connectable:false acceptable:false
readable:true writeable:true
- >TEST>Selection Key.  connectable:false acceptable:false
readable:false writeable:true
- Try to read…
- >TEST>Selection Key.  connectable:false acceptable:false
readable:false writeable:true
- Message Received:CLIENT_ID:Test Client TS_REQ.
- >TEST>Selection Key.  connectable:false acceptable:false
readable:false writeable:true
- >TEST>Selection Key.  connectable:false acceptable:false
readable:false writeable:true
- Message To send:CLIENT_ID:Test Client TS_RESP:Tue Sep 25
18:45:33 CEST 2012
- >TEST>Selection Key.  connectable:false acceptable:false
readable:false writeable:true
- Try to send a message…
- >TEST>Selection Key.  connectable:false acceptable:false
readable:false writeable:true
- >TEST>Selection Key.  connectable:false acceptable:false
readable:true writeable:true
- >TEST>Reading incoming message…
```

```
- >TEST>Read message:CLIENT_ID:Test Client TS_RESP:Tue Sep 25
18:45:33 CEST 2012
- >TEST>Closing channel
- >TEST>Closing the test
- Request to shut-down
- Out of the select wait…
- Server no longer running
```

Da una rapida analisi del file di log si evidenzia il seguente comportamento:

1. Viene eseguito il test che per prima cosa si occupa di lanciare il server e quindi effettua il collegamento con lo stesso
2. Come risposta a questa richiesta sul server arriva un evento di accettazione (**acceptable:true**)
3. Il server esegue l'accettazione della richiesta
4. Il test riceve un evento di avvenuta accettazione e quindi è abilitato ad inviare messaggi (**writeable:true**)
5. Il test scrive un messaggio al server, il quale genera sul server un evento di lettura (**readable:true**)
6. Il server legge il messaggio (**Message Received:CLIENT_ID:Test Client TS_REQ.**) comprende che si tratta di una richiesta valida e quindi risponde (**Message To send:CLIENT_ID:Test Client TS_RESP:Tue Sep 25 18:45:33 CEST 2012**)
7. Il test riceve notifica della presenza di un evento di lettura (**readable:true**)
8. Legge il messaggio e quindi termina.

Conclusioni

Questo capitolo è stato dedicato alla presentazione di una breve descrizione del framework NIO al fine di fornire al lettore gli strumenti necessari per una piena comprensione del Capitolo 4 su NIO2.

Le caratteristiche principali di NIO sono operazioni di I/O asincrone e quindi non bloccanti, possibilità di eseguire il mapping in memoria dei file ed operazioni orientate ai blocchi che sfruttano i servizi messi a disposizione dalle piattaforme di esecuzione. Il framework è basato sui seguenti tre concetti fondamentali: buffer (sono contenitori di dati la cui implementazione è una specializzazione della classe astratta java.nio.Buffer), channel (ossia canali di vario tipo che rappresentano le connessioni con dispositivi di vario tipo in grado di eseguire operazioni I/O) e selector (che insieme ai canali permettono di definire operazioni multiplex e non-bloccanti. Multiplex significa che uno stresso Thread è in grado

di gestire diversi canali). Gli elementi fondamentale del framework NIO sono illustrati per mezzo di un esempio Server NIO/Client NIO. La lettura di questa appendice fornisce tutti gli elementi per comprendere sia le origini, sia l'evoluzione del framework NIO.

Appendice C
Visitor pattern

Prefazione

Questa appendice è dedicata alla presentazione del pattern del visitatore (Visitor Pattern, cfr [Gamma et al, 1994]). Si tratta di un pattern del gruppo dei "comportamentali" (behavioural). In questo gruppo appartengono i pattern che si occupano di algoritmi o meglio della loro gestione, di relazioni e responsabilità tra oggetti. Così come per gli altri pattern, esiste una nutritissima letteratura che ne spiega vantaggi, scenari ideali di utilizzi, struttura, possibili problemi, impieghi inopportuni e così via. In questo capitolo intendiamo darne una lettura molto pratica e assolutamente mirata al motivo per cui è stato introdotto in questo libro: **fornire informazioni di background per la piena comprensione della feature WalkFileTree** introdotta con Java SE 7 (cfr. capitolo 4 "NIO 2"). A tal fine viene presentato un esempio relativo al file system.

Prima di avviare la discussione, si vuole presentare un semplice esempio per far comprendere fin da subito il modo operandi di questo pattern. L'esempio è attinto al mondo dei trasporti e più precisamente a quello dei Taxi e ha inizio con una persona (il cliente) che prenota un taxi, il quale in questo contesto "recita" il ruolo del visitatore. Il taxi una volta giunto al punto di prelievo (per esempio il portone di casa) notifica al cliente il suo arrivo (richiede di accettare la visita), e una volta che il cliente accetta la visita e siede a bordo del taxi, quest'ultimo assume la responsabilità di portare la persona a destinazione (esegue la feature della visita), magari chiedendo, di tanto in tanto, qualche informazione al cliente.

Il problema classico

L'obiettivo primario del pattern del visitatore consiste nel poter rappresentare in modo coeso una funzionalità da eseguire su una serie di elementi di un predefinito grafo di oggetti. Uno dei grandi vantaggi offerti dal pattern consiste nel permettere di definire nuove operazioni (o di aggiornare esistenti) senza dover cambiare le classi che rappresentano gli elementi sui quali le operazioni devono essere eseguite.

Si consideri una classica struttura di un file system. In questo caso la relativa struttura prevede essenzialmente due tipi di nodi concreti: file e directory e su questa struttura si vuole poter eseguire una serie di operazioni, come per esempio determinare l'occupazione totale di memoria a partire da un dato percorso, dar luogo ad un rappresentazione grafica di una directory, e così via.

La strategia classicamente utilizzata nel mondo O.O. per risolvere problemi di questo tipo consiste nel definire l'implementazione dei necessari metodi direttamente in tutti gli elementi nel grafo degli oggetti. Sebbene in questo caso non si tratterebbe di un grande onere visto che esistono solo due (o tre classi considerando anche la classe astratta), esistono altri scenari dove la situazione è molto più evidente. Si consideri il caso di un parser il quale una volta terminata la prima fase (tokenizzazione) genera un grafo di nodi che rappresentano i vari elementi: parole chiave, dichiarazione di variabili, assegnamento di variabili, espressioni matematiche, metodi, etc. Su ciascuno di questi si deve poter eseguire una serie di funzioni, come per esempio: verificare che tutte le variabili sia state definite, controllare che le variabili siano inizializzate prima del relativo utilizzo, type check, e così via. Pertanto, si genera una struttura come quella mostrata nel diagramma delle classi di figura 1.

Questo approccio, come al solito, offre alcuni vantaggi ma anche tutta una serie di importanti svantaggi, come per esempio:

- l'aggiunta di una nuova feature chiedere di cambiare tutte le classi che rappresentano la struttura;
- la rappresentazione delle varie feature si ottiene dalla collaborazione di metodi presenti in tutte le classi della struttura. Ciò può facilmente generare confusione, rendere il codice meno leggibile (una stessa funzione è organizzata in diversi oggetti) e quindi meno manutenibile.

Una soluzione alternativa consiste nell'incapsulare l'implementazione delle varie feature desiderate in apposite classi, specializzazioni dell'interfaccia visitatore (**Visitor**, cfr. figura 2).

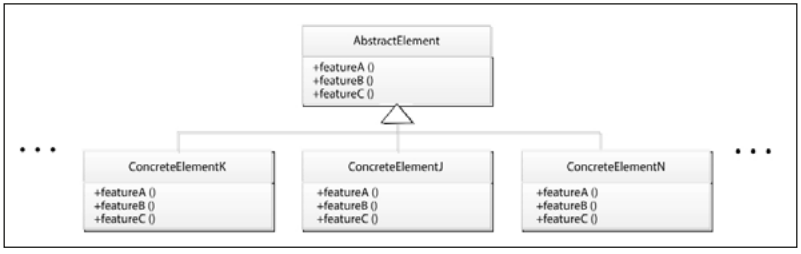


Figura 1. *Implementazione di una serie di feature su un grafo di oggetti.*

Il nome è dovuto al fatto che le istanze di questo tipo hanno il compito di attraversare (visitare) gli elementi del grafo di oggetti (la struttura del file system). Si tratta di visitatori ben educati che chiedono il permesso di poter far visita. Quando

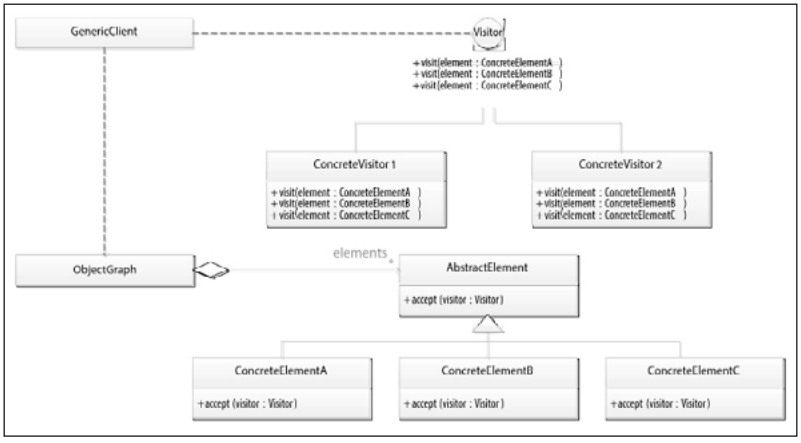


Figura 2. *Diagramma delle classi del visitor pattern in forma canonica.*

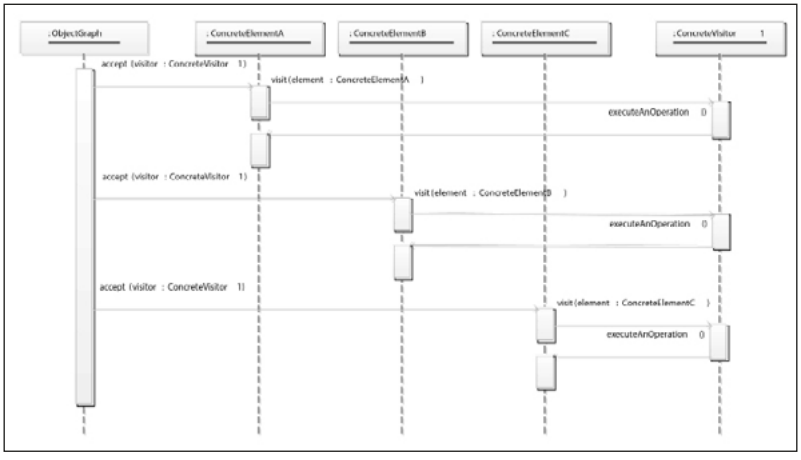


Figura 3. *Sequence diagram del visitor patten.*

un elemento della struttura accetta (metodo accept) la visita di un visitatore, invoca il metodo del visitatore (visit) che riceve come parametro formale un elemento del proprio tipo. Il visitatore quindi può eseguire la determinata feature sullo specifico nodo (cfr. Figura 3). Questo è in buona sostanza una semplificazione del metodo che secondo la strategia iniziale ero allocato nel nodo dati anziché nel Visitor.

La doppia consegna (double-dispatch)

Come visto una la caratteristica centrale del pattern del visitatore consiste nel concentrare l'implementazione di una determinata feature in una sola classe il che permette di aggiungere nuove operazioni senza dover modificare le classi dati. La tecnica utilizzata si chiama della doppia consegna (double-dispatch) in contrapposizione con la singola consegna (single-dispacth) che altro non è se non l'applicazione diretta del principio del polimorfismo. Vi è una classe (tipicamente astratta) che implementa un metodo che poi viene ridefinito (overridden) da opportune sotto-classi. Pertanto, quale metodo sia poi effettivamente eseguito dipende dallo specifico tipo dell'oggetto che si sta utilizzando. Esempio: vi è la classe astratta AbstractShape che definisce il metodo astratto draw. Questa classe è poi specializzata da una serie di classi concrete, come per esempio Square, Triangle, Circle, etc. Il sistema tratta, per quanto possibile, oggetti "shape" in modo generico: come se fossero istanze della classe AbstractShape. Pertanto, quando un componente di rendering grafico invoca il metodo draw di un oggetto "shape", in effetti, viene eseguito il metodo definito dal tipo concreto di tale oggetto (Square, Triangle, Circle, e così via). Pertanto, nella singola consegna, la determinazione del metodo eseguito dipende dal metodo invocato e dello specifico tipo dell'istanza.

La doppia consegna è molto simile solo che in questo casso sono coinvolte due applicazioni del principio del polimorfismo. Nel pattern del visitatore ciò è dovuto alla presenza di due gerarchie: quella degli elementi da visitare e quella dei visitatori. Quindi, come evidenziato nelle figure 2 e 3 (cfr.), vi è una prima applicazione del polimorfismo durante l'accettazione della visita (metodo accept, gerarchia dei dati) ed una seconda per eseguire il conseguente lavoro (metodo visit, nella gerarchia dei visitatori). Ciò fa sì che l'esecuzione finale dipende dal metodo invocato (il nome della richiesta iniziale) a dai due tipi concreti dei "ricevitori" delle invocazioni.

Esempio

È ora il momento di mostrare l'esempio concreto (cfr. figura 4).

Si tratta di un semplice Visitor che si occupa di determinare l'occupazione di memoria di una data directory o file. Le informazioni relative a directory e file

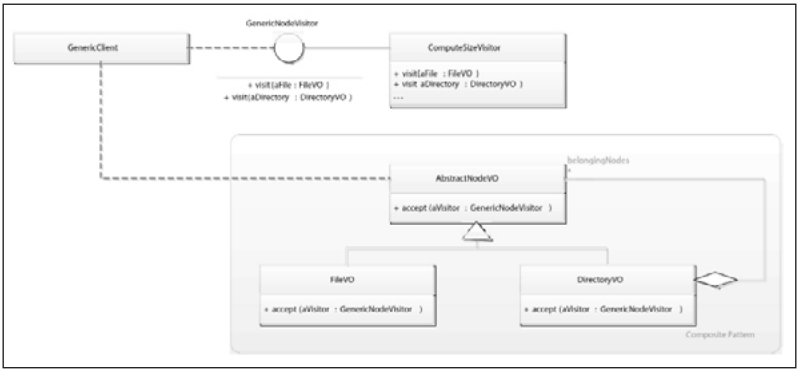


Figura 4. *Class diagram dell'esempio proposto.*

(che a dire il vero sono molto povere per far sì che il lettore possa concentrarsi meglio sull'implementazione del pattern) sono memorizzate in un grafo di oggetti (Value Objects, ossia oggetti che hanno come unico compito quello di veicolare i dati) implementato secondo le direttive del composite pattern. Questo permette di rappresentare efficacemente una struttura intrinsecamente ricorsiva atta a dar luogo ad alberi di oggetti.

In particolare, ogni directory può includere una serie dei nodi che possono essere:

- file, e si termina la ricorsione in quanto un file non può essere ulteriormente decomposto; pertanto rappresenta il passo base della ricorsione, quello che la termina;
- directory, che come tale può contenere altri file e directory, e quindi rappresenta il passo ricorsivo.

In questo contesto è stato definito un solo Visitor concreto, però, ovviamente, se ne potrebbero definire altri, come per esempio una feature per disegnare il una directory attraverso un widget ad albero da mostrare sullo schermo, una atta ad individuare opportuni elementi in qualsiasi grado di profondità, etc.

Un importante elemento da evidenziare è che sebbene l'implementazione della feature richieda di attraversare il grafo di oggetti DirectoryVO e FileVO e che questi siano astratti dalla classe AbstractNodeVO (una directory include un insieme di nodi dichiarati come private AbstractNodeVO[] belongingNodes), **non è mai necessario dar luogo agli odiatissimi down casting**. L'applicazione del pattern del visitatore permette di rimuovere la necessità di ricorrere a questa sozzura.

```
public abstract class AbstractNodeVO {

  // -------------- CONSTANTS SECTION --------------

  // -------------- ATTRIBUTES SECTION --------------

  /** element path (without name) */
  private String path;

  /** element name */
  private String name;

  // --------------- METHODS SECTION ---------------
  // ----------------------------- constructors

  /**
   * basic constructor
   */
  public AbstractNodeVO() {
  }

  /**
   * Full constructor
   * @param path   path without name
   * @param name   name
   */
  public AbstractNodeVO(String path, String name) {
    this.path = path;
    this.name = name;
  }

  // ------------------------ getters

  /**
   * @return the path
   */
  public String getPath() {
    return path;
```

```
}

/**
 * @return the name
 */
public String getName() {
  return name;
}

// --------------- Visitor pattern methods

/**
 * accept the visit of “visitor” pattern element
 * and in turns, it requires to be visited
 */
public abstract void accept(
        GenericNodeVisitor aVisitor);


// ------------------ standard object methods

/*
 * (non-Javadoc)
 * @see java.lang.Object#hashCode()
 */
@Override
public int hashCode() {
  . . .
}

/*
 * (non-Javadoc)
 * @see java.lang.Object#equals(java.lang.Object)
 */
@Override
public boolean equals(Object obj) {
  . . .
}
```

```
    /*
     * (non-Javadoc)
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString() {
      . . .
    }


}  // --- end of class ---
```

Listato 1. *Classe astratta AbstractNodeVO*

```
public class FileVO extends AbstractNodeVO {

  // -------------- CONSTANTS SECTION ---------------

  // -------------- ATTRIBUTES SECTION --------------

  /** file size */
  private long size;

  /** indicates whether the file is executable or not */
  private boolean executable;

  // --------------- METHODS SECTION ---------------

  // ----------------------------- constructors

  /**
   * basic constructor
   */
  public FileVO() {
  }

  /**
```

```
     * Full constructor
     * @param path       path without name
     * @param name       file name
     * @param size       file size
     * @param executable true is an executable file
     *                   false otherwise
     */
    public FileVO(
            String path,
            String name,
            long size,
            boolean exutable) {
      super(path, name);
      this.size = size;
      this.executable = exutable;
    }

    // ---------------------------- getters

    /**
     * @return the size
     */
    public long getSize() {
      return this.size;
    }

    /**
     * @return true is executable, false otherwise
     */
    public boolean isExecutable() {
      return executable;
    }

    // --------------------------- visitor pattern method

    @Override
    public void accept(GenericNodeVisitor aVisitor) {
      if (aVisitor != null) {
```

```
      aVisitor.visit(this);
    }
  }


  // ---------------------------- standard object methods

  /*
   * (non-Javadoc)
   * @see java.lang.Object#hashCode()
   */
  @Override
  public int hashCode() {
    . . .
  }

  /*
   * (non-Javadoc)
   * @see java.lang.Object#equals(java.lang.Object)
   */
  @Override
  public boolean equals(Object obj) {
    . . .
  }

  /* (non-Javadoc)
   * @see java.lang.Object#toString()
   */
  @Override
  public String toString() {
    . . .
  }

}  // --- end of class ---
```

Listato 2. Classe FileVO

```
public class DirectoryVO extends AbstractNodeVO {
```

```
// -------------- CONSTANTS SECTION --------------

// -------------- ATTRIBUTES SECTION -------------

/** list of belonging nodes (file and directories) */
private AbstractNodeVO[] belongingNodes;

// --------------- METHODS SECTION ---------------

// ----------------------------- constructors

/**
 * basic constructor
 */
public DirectoryVO() {
}

/**
 * Full constructor
 * @param path           path without name
 * @param name           file name
 * @param belongingNodes list of belonging nodes
 */
public DirectoryVO(
        String path,
        AbstractNodeVO[] belongingNodes) {
  super(path, null);
  this.belongingNodes = belongingNodes;
}

// ---------------------------------- getters

/**
 * @return the list of belonging nodes
 */
public AbstractNodeVO[] getBelongingNodes() {
  return this.belongingNodes;
```

```
}

// -------------------- visitor pattern method

@Override
public void accept(GenericNodeVisitor aVisitor) {
  if (aVisitor != null) {
    aVisitor.visit(this);
  }
}

// ---------------- standard object methods

/*
 * (non-Javadoc)
 * @see java.lang.Object#hashCode()
 */
@Override
public int hashCode() {
  . . .
}

/*
 * (non-Javadoc)
 * @see java.lang.Object#equals(java.lang.Object)
 */
@Override
public boolean equals(Object obj) {
  . . .
}

/*
 * (non-Javadoc)
 * @see java.lang.Object#toString()
 */
@Override
public String toString() {
  . . .
```

```
    }

  }  // --- end of class ---
```

Listato 3. *Classe DirectoryVO.*

```
  public interface GenericNodeVisitor {

    /**
     * Visit the given file
     */
    public void visit(FileVO aFile);


    /**
     * Visit the given directory
     */
    public void visit(DirectoryVO aDirectory);


  }  // --- end of interface ---
```

Listato 4. *Interfaccia GenericNodeVisitor*

```
  public class ComputeSizeVisitor
         implements GenericNodeVisitor {

    // -------------- CONSTANTS SECTION ---------------

    private static Logger LOGGER =
          Logger.getLogger(ComputeSizeVisitor.class);

    // -------------- ATTRIBUTES SECTION --------------

    /** total size */
    private long totalSize;
```

```
// ---------------- METHODS SECTION ----------------

/**
 * Default constructor
 */
public ComputeSizeVisitor() {

}

// -------------------------------- visitor methods

@Override
public void visit(FileVO aFile) {
  LOGGER.info("Visiting file:"+aFile.toString());

  if (aFile != null) {
    totalSize += aFile.getSize();
  }
}


@Override
public void visit(DirectoryVO aDirectory) {

  LOGGER.info("Visiting directory:"+
                aDirectory.toString());

  if (aDirectory != null) {
    AbstractNodeVO[] belongingNodes =
            aDirectory.getBelongingNodes();

    if ( (belongingNodes != null) &&
         (belongingNodes.length > 0) ) {
      for (AbstractNodeVO currNodeVO : belongingNodes) {
        currNodeVO.accept(this);
      }
    }
  }
```

```
}


// ---------------------------------- getter methods

/**
 * @return the total Size
 */
public long getTotalSize() {
  return totalSize;
}

// ---------------------------------- object methods

/*
 * (non-Javadoc)
 * @see java.lang.Object#hashCode()
 */
@Override
public int hashCode() {
  . . .
}


/*
 * (non-Javadoc)
 * @see java.lang.Object#equals(java.lang.Object)
 */
@Override
public boolean equals(Object obj) {
  . . .
}


/*
 * (non-Javadoc)
 * @see java.lang.Object#toString()
 */
```

```
    @Override
    public String toString() {
      . . .
    }


}  // --- end of class ---
```

Listato 5. *Classe ComputeSizeVisitor.*

```
public class ComputeSizeVisitorTest {

  // -------------- CONSTANTS SECTION ---------------

  // -------------- ATTRIBUTES SECTION --------------

  /** directory used for the tests */
  private DirectoryVO dir;

  // --------------- METHODS SECTION ----------------


  /**
   * Creates the structure to use for the test
   */
  @Before
  public void setup() {

    dir = new DirectoryVO(
        "C:\\",
        new AbstractNodeVO[] {
          new FileVO("C:\\", "Readme.txt", 10_000, false),
          new DirectoryVO(
            "C:\\prj",
            new AbstractNodeVO[] {
              new FileVO("C:\\prj\\", "myDoc.doc", 145_000, false),
              new FileVO("C:\\prj\\", "myText.txt", 45_000, false)
```

```
            }
          )
        }
      );
  }


  @Test  // positive test
  public void testComputeSize() {
    ComputeSizeVisitor computeSizeVisitor =
                new ComputeSizeVisitor();


    computeSizeVisitor.visit(dir);

    assertTrue(computeSizeVisitor.getTotalSize() == 200_000);

  }

}  // --- end of class ---
```

Listato 6. *Classe ComputeSizeVisitorTest.*

Applicazioni del pattern

Da quanto descritto in precedenza ne segue che gli scenari ideali per l'applicazione del pattern del visitatore sono:

- la necessità di dover implementare diverse feature su un medesimo grafo di oggetti. In particolare, in questi casi l'applicazione del pattern del visitatore è particolarmente consigliata nei casi in cui si volesse evitare di distribuire l'implementazione della medesima feature in *n*-classi e l'inquinamento del codice degli oggetti della struttura.
- la struttura del grafo di oggetti è ben definita e tende a non variare, mentre l'implementazione delle feature e le feature stesse presentano una maggiore volatilità. Il caso opposto invece evidenzia delle criticità del pattern del Visitor.
- quando le classi che definiscono gli elementi del grafo degli oggetti implementano differenti interfacce (e quindi sono di tipo diverso) e si ha la necessità di

implementare feature molto dipendenti dalle specifiche classi. In questo caso **l'applicazione del Visitor pattern permette di evitare diversi odiosissimi down casting**.

Pro et contra

I vantaggi dell'applicazione del pattern del visitatore sono:

- immediatezza e semplicità relative all'implementazione e modifica delle feature;
- concentrazione dell'implementazione delle varie feature. Queste non sono più "disperse" nelle varie classi del grafo dei dati ma sono concentrate negli appositi visitatori;
- semplificazione delle implementazioni stateful. Il visitatore permette di avere uno stato in cui accumulare i valori desiderati (come per esempio la dimensione dei file) man mano che si procede con la visita dei nodi. In mancanza del pattern del visitatore, questo dovrebbe essere inserito esplicitamente come parametro formale in tutte le operazioni, distribuite nei vari nodi, che implementano la feature, anche in quelli che non prevedono alcun contributo allo stato;
- l'implementazione del Visitor pattern su grafi di oggetti non banali, permette di evitare la fastidiosa procedura del downcasting. Tipicamente, la navigazione di grafi di oggetti non banali in cui sono presenti delle classi astratte che accumunano il comportamento condiviso di una serie di oggetti (come per esempio AbstractNodeVO) richiede di eseguire test del tipo "if (currentObject instanceof ObjectType)" seguite dal down-cast (ObjectType)currentObject;".

Insiemi a questi vantaggi, come lecito attendersi, esistono una serie di svantaggi, ed in particolare:

- lo scenario in cui le feature sono abbastanza stabili mentre la struttura è volatile crea maggiori problemi. Per esempio, l'aggiunta di un nuovo elemento al grafo di oggetti richiede di modificare l'implementazione di tutti di visitatori;
- l'interfaccia degli elementi del grafo di oggetti deve essere abbastanza potente giacché il Visitor dovrà utilizzare tali metodi per implementare la propria feature. Questo, in casi non molto frequenti, può portare a rendere pubblici metodi che altresì non lo sarebbero.

Conclusioni

Questo capitolo è stato dedicato ad una breve presentazione del pattern del visitatore (Visitor) al fine di fornire ai lettori una chiave di lettura molto pratica

della feature WalkFileTree introdotta con NIO 2. In particolare, questo pattern prevede di incapsulare l'implementazione di un'intera feature, la cui esecuzione richiede di navigare un grafo di oggetti, in una sola classe. Questa strategia si contrappone all'approccio OO classico che prevede di ottenere simili feature attraverso la collaborazione di tutti i nodi del grafo da navigare.

L'implementazione di feature basate su questo pattern fa sì che sia abbastanza facile ed immediato modificare o aggiungere nuove feature, di rendere il codice (soprattutto quello della feature), più facile da leggere e quindi manutenibile, di ottenere implementazioni in grado di mantenere uno stato e molto importante, di evitare noiosissimi down casting.

Come al solito non è tutto ora ciò che luccica, quindi, quando si applica il pattern del Visitor bisogna porre attenzione ad alcuni elementi. In particolare, in presenza di diverse feature e qualora la struttura presenti una certa volatilità, l'applicazione del pattern del Visitor finirebbe per richiedere continue modifiche dei vari visitatori. Inoltre, è necessario assicurarsi che gli elementi del grafo dei dati siano in grado di fornire tutti i dati di cui ha bisogno il visitatore. In alcuni casi ciò può richiedere di rendere pubblici alcuni metodi che altrimenti sarebbero privati, in altre parole può richiedere di esporre più comportamento di quello che si vorrebbe. Il Visitor è un pattern molto potente ed elegante, ma come tutti gli strumenti deve essere utilizzato per risolvere problemi per cui è stato disegnato.

Appendice D
Stile funzionale in un chicco di grano

Prefazione

La programmazione funzionale è un importante paradigma che ha ispirato diversi linguaggi di programmazione, basti citare LISP e Scala, che da tempo appassionano la comunità informatica ed animano dibattiti tra sostenitori dei vari paradigmi. La logica conseguenza è che esista una nutritissima letteratura dedicata all'argomento. Nei seguenti paragrafi è riportata una breve introduzione al solo scopo di fornire i lettori con una breve introduzione sulla programmazione funzionale al fine di permettere una migliore comprensione del progetto Lambda, delle relative sfide e del suo impatto sulla piattaforma Java. Pertanto questa breve presentazione è orientata a fornire i principi e conoscenze necessarie per una piena comprensione delle espressioni Lambda.

Breve storia

La programmazione funzionale deve le proprie origini al Lambda calcolo che può essere considerato a tutti gli effetti il primo linguaggio di programmazione funzionale quantunque non fu progettato per essere eseguito da calcolatori. Si tratta di un modello computazionale progettato da Alonzo Church ([Church, 1956]) negli anni trenta che fornisce un metodo molto formale di descrivere il processo di valutazione di funzioni. Il primo linguaggio di programmazione funzionale moderno, fu l'IPL (Information Processing Language, linguaggio di elaborazione informazioni/dati), sviluppato da Newell, Shaw e Simon a metà degli anni cinquanta. Ma **il grande successo della programmazione funzionale si deve senza dubbio al LISP (LISt Processing), sviluppato alla fine degli anni Cinquanta da John McCarthy** ([McCarthy, 1958]), durante la sua collaborazione con il MIT (Massachusetts Institute of Technology), per computer della serie IBM 700/7000. Il linguaggio Scheme fu un successivo tentativo di semplificare e migliorare il LISP. Negli anni settanta, all'Università di Edimburgo fu creato il linguaggio ML e all'Università di Kent David Turner creò il linguaggio Miranda. Il linguaggio Haskell fu creato e rilasciato alla fine degli anni ottanta come tentativo di mettere insieme molte delle idee della ricerca sulla programmazione

funzionale. Allo stato attuale la programmazione funzionale è tornata in auge grazie a linguaggi come Scala, Clojure e F#. Nel mondo Java c'è una grande fermento atto ad integrare aspetti funzionali nel paradigma OO.

Qualche nozione di programmazione funzionale

La programmazione funzionale è uno stile di programmazione (paradigma) in cui gli algoritmi sono implementati in termini di una serie di funzioni. Ciò fa sì che il flusso di esecuzione si risolva in una serie di valutazioni di espressioni matematiche. Pertanto il lavoro dello sviluppatore è implementare algoritmi in termini di una funzione, normalmente espressa come una funzione matematica. Questa, secondo i buoni principi del diviti et impera, è suddivisa in un certo numero di funzioni "ausiliarie". L'enfasi, quindi, è posta sulla definizione di funzioni. Ciò è in contrasto con gli altri stili di programmazione, come quello strutturato ed imperativo in cui i programmi sono implementati attraverso una serie di dichiarazioni il cui obiettivo consiste nel cambiare lo stato di singoli moduli/oggetti e quindi, in ultima analisi, del sistema. **La programmazione funzionale tende ad utilizzare stati immutabili**: i risultati non sono ottenuti per mezzo di un cambiamento di stato del programma, ma costruendo nuovi stati a partire dai precedenti (un po' come avviene per le stringhe Java: invece di variare una data stringa gli aggiornamenti avvengono creando nuovi oggetti stringa). Ciò rimuove alla base la presenza l'insorgenza di **effetti collaterali** (**side effect**) e pone le basi per una programmazione parallela semplificata: dal momento che i valori originali non possono essere variati questi possono essere tranquillamente condivisi da più Thread che a loro volta generano nuovi valori, senza correre il rischio di generare race conditions. Chiaramente, poi è necessario disporre di primitive che consentono di aggregare i risultati in modo thread-safe.

La programmazione funzionale prevede che le funzioni siano "cittadini di prima classe" (first-class). La nozione di "cittadino di prima classe" o "di prima classe di elementi" nei linguaggio di programmazione fu introdotto dal ricercatore inglese Christopher Strachey nel 1960 proprio nel contesto della programmazione funzionale. Mentre per quanto riguarda una sua formulazione rigorosa bisogna attendere il libro "Structure and Interpretation of Computer Programs" di Gerald Jay Sussman e Harry Abelson ([Abelson et al, 1984]) utilizzato come libro di testo presso il MIT. Semplificando, **un'entità di un linguaggio di programmazione è un cittadino di prima classe quando le seguenti condizioni sono soddisfatte:**

- possono essere assegnate a variabili;
- possono essere passati come argomenti ai metodi;

- possono essere restituiti come risultati dei metodi;
- possono essere incluse in strutture di dati.

Semplificando al massimo, la logica conseguenza è che elemento di prima classe possono essere trattati come tipi primitivi o un qualsiasi oggetto in un linguaggio OO, il che, nel caso delle funzioni non è una meccanismo così immediata: basti pensare alla possibilità di poter manipolare una funzione all'interno di un'altra e restituirne come risultato un'altra ancora. In questo caso si parla di **HOFs (Higher-Order Functions, funzioni di ordine superiore)**, ossia funzioni che prevedono altre funzioni come argomento di input e/o di output. In matematica l'esempio classico di funzioni di ordine superiore sono l'operatore differenziale: si tratta di una funzione che esegue il mapping di una funzione di input con una di output (per esempio la derivata della prima della funzione *f(x) = x^2* è *f'(x) = 2x*). In termini programmativi, un semplice esempio di HOF è ottenibile per mezzo delle funzioni predefinite map in Haskell e mapcar in LISP che prevedono come input una funzione ed una lista. Ciò può essere utilizzato per generare una nuova lista i cui valori sono i risultati dell'esecuzione della funzione su ogni elemento della lista. Per esempio, la seguente dichiarazione permette di definire una funzione (`x – y`) che sottrae il valore y, definito come secondo parametro, a tutti gli elementi della lista definita come primo parametro:

```
subtractOneFromList l y = map (\x -> x - y) l
```

Da notare che la funzione (`\x –> x – y`) fornita come parametro a map realizza una "chiusura" (closure) giacché fa riferimento ad una variabile (y) al di fuori della propria definizione (body). In generale, **si ha una chiusura quando una funzione fa riferimento ad una variabile non vincolata e pertanto definita libera (free variable)** come riportato nel listato di seguito.

```
int value = 5;

UnaryOperator<Integer> addValue = x -> x + value;
```

Nell'esempio riportato sopra, la funzione **AddValue** fa riferimento, utilizza, la variabile **value** che appartiene all'ambito (scope) che l'ha include.

Tipicamente questo concetto si applica quando una funzione fa riferimento a degli elementi definiti dalla funzione esterna che l'ha include. Da notare che, comunemente, una variabile è libera non in modo assoluta, ma in riferimento al contesto di una determinata espressione.

Le funzioni possono avere dei nomi (**subtractOneFromList**), come in ogni altro linguaggio, o essere definite anonimamente (a volte anche a run-time) usando l'astrazione lambda, ed essere usate come valori in altre funzioni.

Un'altra interessante caratteristica delle funzioni da considerare è il concetto di **Currying** (il nome deriva dal nome dell'ideatore Haskell Curry che lo ideò indipendentemente ed in parallelo con Moses Schönfinkel) che rappresenta la possibilità di concatenare funzioni. Si tratta della tecnica che permette di trasformare una funzione che accetta più argomenti (*n*-upla di argomenti) in modo tale che esso possa essere definito come una catena di funzioni ciascuno con un singolo argomento. Si consideri il caso di avere una funzione del tipo

```
f(x, y, z)
```

dove sia il dominio sia il codominio e l'insieme dei numeri interi

```
( (int, int, int) -> int )
```

Questo principio, espresso in forma ingegneristica e non matematica, permette di trasformare tale funzione in un'altra avente la seguente forma:

```
g(x)(y)(z) ( int -> (int -> (int -> int)) )
```

che da un punto di vista pratico consiste nel poter concatenare funzioni.

Le HOF, inoltre, si prestano per definire iterazione speciali, come cicli paralleli.

I linguaggi di programmazione funzionale si suddividono in puri e non. I primi sono ispirati ai linguaggi matematici e sono caratterizzati dalle seguenti proprietà:

- non permettono effetti collaterali;
- i vari algoritmi sono implementati esclusivamente attraverso l'applicazione di funzioni ad argomenti;
- funzioni complesse sono definite ricorrendo a funzioni più semplici attraverso il ricorso ad espressioni condizionali e alla ricorsione.

Quest'ultima è la forma canonica, e spesso l'unica, per eseguire iterazioni come scorrere insiemi di elementi. Da notare che implementazioni del linguaggio funzionali naturalmente includono delle ottimizzazioni per garantire che un utilizzo massiccio della ricorsione non crei problemi di memoria e performance.

Queste caratteristiche permetto di raggiungere l'importante proprietà della **trasparenza referenziale: il risultato della valutazione di un'espressione è indipendente da altre parti del programma**. Ciò equivale ad asserire che l'applicazione di una funzione agli stessi argomenti produce sempre lo stesso risultato. Chiaramente si tratta di una proprietà fondamentale di tutti i concetti matematici ed in particolare delle funzioni.

Da notare che giacché la valutazione delle espressioni di linguaggi funzionali gode della proprietà della trasparenza referenziale, ne segue che è possibile essere valutare una data espressione in qualsiasi momento ed ottenere lo stesso risultato. **Ciò fa sì che sia possibile rinviare il calcolo dei valori al momento del loro effettivo utilizzo: si può eseguire la lazy evaluation**. Tale valutazione pigra evita inutili calcoli e permette di definire ed utilizzare strutture infinite. Sebbene disporre di linguaggi funzionali puri offre una serie di interessanti vantaggi, esistono degli scenari in cui ciò non è sempre possibile. Per esempio, si consideri la necessità di utilizzare in maniera efficiente matrici mutevoli o operazionali I/O. Per consentire ai vari linguaggi funzionali di rimanere relativamente puri, si fa sì che il core resti tali e si permette di avere delle estensioni non pure.

Queste caratteristiche tra gli altri vantaggi fanno sì che la programmazione funzionale supporti il processo di refactoring del codice, spesso l'implementazione è più vicina alla specifica, permette di dar luogo ad implementazioni concise riducendo il codice accessorio (**boilerplate**) e ripetizioni. Molti cicli, per esempio, possono essere espressi utilizzando le mappe (come visto sopra) ed il concetto di **folds** (si tratta di una famiglia di funzioni HOF che si occupa di eseguire delle elaborazioni su strutture di dati secondo qualche ordine e di generare dei valori di ritorno). Ultimo ma non ultimo, la programmazione funzionale tende naturalmente alla programmazione parallela.

Conclusione

L'obiettivo del presente capitolo è fornire il lettore con alcune nozioni base sulla programmazione funzionale utili per comprendere appieno le espressioni Lambda. Una trattazione approfondita richiederebbe un intero volume.

La programmazione funzionale è un paradigma in cui gli algoritmi sono implementati in termini di una serie di funzioni. Quindi le funzioni sono l'entità centrale e come tale sono "cittadini di prima classe" (first-class). Ciò significa che le funzioni possono essere assegnate a variabili, passate come argomenti ai metodi, restituite come risultati dei metodi e possono essere incluse in strutture di dati. Questo permette di implementare un concetto molto potente definito "Code as data".

La programmazione funzionale offre molti vantaggi, tra i quali i più importanti sono il supporto al processo di refactoring del codice, la chiarezza dell'implementazione che tende ad essere più vicina alla specifica, la concisione del codice grazie alla minimizzazione del codice accessorio (**boilerplate**) e ripetizioni. Inoltre, poiché la programmazione funzionale tende ad utilizzare stati immutabili, viene rimossa alla base la presenza dell'insorgenza di effetti collaterali (**side effect**) e pone le basi per una programmazione parallela semplificata.

Riferimenti bibliografici

[Abelson et al, 1984]

Harold Abelson, Gerald Jay Sussman, Julie Sussman, *Structure and Interpretation of Computer Programs (SICP)*, MIT Press written by Massachusetts Institute of Technology, 1984

[Amdahl, 1967]

Gene M Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, IBM SunnyvaleCalifornia, AFIPS spring joint computer conference, 1967

`http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf`

[Asanovic et al, 2006]

K. Asanovic et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, tech. report UCB/EECS- 2006-183, Dept. Electrical Eng. and Computer Science, Univ. of Calif., Berkeley, 2006.

[Church, 1956]

Alonzo Church, *Introduction to Mathematical Logic*, Princeton University Press (Princeton, NJ: 1944, 1956). ISBN 0-691-02906-7

[Darcy, 2009]

Joe Darcy, 2009 Proposals TOC, selezione finale delle proposte di modifica da far confluire in Java SE 7

`https://wikis.oracle.com/display/ProjectCoin/2009ProposalsTOC`

[Darcy, 2011]

Joseph D. Darcy, *JEP 120: Repeating Annotations*

`http://openjdk.java.net/jeps/120`

[Gamma et al, 1994]

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design patterns: elements of reusable object-oriented software*, 1994

[Goetz, 2011]
Brian Goetz, *A peek past Lambda*, 2011
http://mail.openjdk.java.net/pipermail/lambda-dev/2011-August/003877.html

[Goetz et al, 2006]
Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea, *Java Concurrency in Practice*, Addison-Wesley, 2006.

[Gosling, 2010]
James Gosling, *Time to move on...*, 9 aprile 2010
http://nighthacks.com/roller/jag/entry/time_to_move_on

[Gustafson, 1988]
J. Gustafson, *Reevaluating Amdahl's Law*, Communications of the ACM, Volume 31, May 1988

[Harned, 2012]
Edward Harned, *A Java™ Fork-Join Calamity Parallel processing with multi-core Java™ applications*, Maggio 2012
http://coopsoft.com/ar/CalamityArticle.html

[Heiss, 2011]
Janice J. Heiss, *Java SE 7 – Plan A or Plan B?*, 20 settembre 2011
https://blogs.oracle.com/javaone/entry/java_se_7_-_plan_a_or_plan_b

[ISO 4217,2008]
ISO 4217:2008, Codes for the representation of currencies and funds
http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=46121

[Java Tut. F.O., 2012]
Java Tutorials, File Operations
http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob

[JDBC 4.1, 2012]
Nuove feature introdotte attaverso JDBC 4.1
http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/jdbc_41.html

[JSR 201, 2004]
JSR 201: "Extending the JavaTM Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import"
http://jcp.org/en/jsr/detail?id=201

[JDK7, 2011]
JDK 7 Milestones
http://openjdk.java.net/projects/jdk7/milestones/

[Knuth, 1998]
Donald Knuth, *The Art of Computer Programming: Volume 3 Sorting and Searching*. ISBN 0-201-89685-0, 1998

[Krill, 2011]
Paul Krill, *Java 7: What's in it for developers. The long-awaited release got off to a rough start but offers a multitude of improvements for developers*, InfoWorld, 24 agosto 2011
http://www.infoworld.com/d/application-development/java-7-whats-in-it-developers-170636

[Lea, 2009]
Doug Lea, *TransferQueue motivation?*
http://cs.oswego.edu/pipermail/concurrency-interest/2009-February/005888.html

[Leiserson, 2010]
Charles E. Leiserson, supervisore del progetto "The Cilk project", MIT CSAIL Supertech Research Group
http://supertech.csail.mit.edu/cilk/

[McCarthy, 1958]
John McCarthy, *Recursive Functions Of Symbolic Expressions And Their Computation By Machine (Part I)*, Communications of the ACM, aprile 1960

[Moore, 1975]
Gordon Moore, *Progress in Digital Integrated Electronics*, IEEE, IEDM Tech Digest (1975) pp. 11-13

[Nimbus, 2012]
Oracle, Nimbus look and feel
http://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/nimbus.html

[OpenJDK] Java Development Kit
http://openjdk.java.net/

[Oracle, 2012]
JSR 335: Lambda Expressions for the Java™, Programming Language, Version 0.5.1. Copyright 2012 Oracle Corporation, 500

[OSGi, 2012]
OSGi Alliance
http://www.osgi.org/Main/HomePage

[Reinhold, 2010]
Mark Reinhold, *There's not a moment to lose! Re-thinking JDK 7*, 08 settembre 2010
http://mreinhold.org/blog/rethinking-jdk7

[Reinhold, 2012]
Mark Reinhold, *There's not a moment to lose! Project Jigsaw: Late for the train*, 17 luglio 2012
http://mreinhold.org/blog/late-for-the-train

[Scherer, 2009]
William N. Scherer III, Doug Lea, Michael L. Scott, *Scalable Synchronous Queues*, "Research Highlights"
http://www.cs.rochester.edu/u/scott/papers/2009_Scherer_CACM_SSQ.pdf

[SCTP, 2004]
Stream Control Transmission Protocol (SCTP)
http://www.sctp.org/

[Smith, 2011]
Daniel Smith, *The Heads and Tails of Project Coin*, Oracle JavaOne, 2011
http://www.oracle.com/javaone/lad-en/session-presentations/corejava/22641-enok-1439101.pdf

[Taft, 2011]
Darryl K. Taft, *Java Creator Gosling Leaves Google for Liquid Robotics*, 30 agosto 2011
http://www.eweek.com/c/a/IT-Management/Java-Creator-Gosling-Leaves-Google-for-Liquid-Robotics-202744/

[Unicode 6, 2011]
The Unicode Consortium. The Unicode Standard, Version 6.0.0, Mountain View, CA: The Unicode Consortium, 2011. ISBN 978-1-936213-01-6
http://www.unicode.org/versions/Unicode6.0.0/

[Vetti Tagliati, 2007]
Luca Vetti Tagliati, *Maven: best practices per il processo di build e di rilascio dei progetti in Java*
http://www2.mokabyte.it/cms/article.run?articleId=S85-L5J-HP3-86O_7f000001_30480431_0844866c

[Vetti Tagliati, 2008]
Luca Vetti Tagliati, *Java Best Practice. I migliori consigli per scrivere codice di qualità*, Edizioni Tecniche Nuove, aprile 2008

[Vetti Tagliati, 2010]
Luca Vetti Tagliati, *Java e le architetture a 64 bit: i puntatori compressi*, MokaByte 157, dicembre 2010
http://www2.mokabyte.it/cms/article.run?articleId=D7E-Q8S-USB-URM_7f000001_24511967_e20206eb